MASTER IN COMPUTER SCIENCE

Bidirectional approach to Channel-based database schema evolution

Beine, Mathieu; Hames, Nicolas

Award date:
2014

Awarding institution:
University of Namur

Université de Namur
Faculté d'informatique
Année académique 2013–2014

Bidirectional approach to Channel-based database schema evolution

Nicolas Hames Mathieu Beine

Maître de stage : Dr Jens WEBER

Promoteur : _____ (Signature pour approbation du dépôt - REE art. 40)
Dr Anthony CLEVE

Mémoire présenté en vue de l'obtention du grade de
Master en Sciences Informatiques

Abstract

Many of today's software applications are backed by database management systems (DBMS), most of them using a relational data model. With increasing system complexity and changing requirements arises the need to adapt and evolve software applications to meet new objectives. In the context of data intensive applications, adaptations may be performed at the database level (e.g., schema changes, data migration) or at the level of the software application (e.g., program code). Changes made at either level often entail changes at the other level in order for the overall system to keep functioning. The synchronization of adaptation at different levels is often referred to as the schema/program co-evolution challenge.

Bidirectional transformations (*bx*) play an important role in the database/program co-evolution challenge. *bx* can be used to decouple the evolution of the database schema from the evolution of the program code, for example, by allowing changes to the database structure to be implemented while some programs can remain unchanged. In earlier work, J. Terwilliger introduced the theoretical concept of a *Channel* as a *bx*-based mechanism to decouple "virtual databases" used by the application code from the actual representation of the data maintained within the DBMS (a.k.a. "physical schema").

In this Master's thesis, we report on considerations and experiments implementing such Channels in practice in the context of a complex real-world application. We focus on Channels implementing Pivot and Unpivot transformations, present different alternatives for generating such Channels and discuss their performance characteristics at scale. We also present a transformational tool to generate these Channels.

Résumé

Aujourd'hui, beaucoup de logiciels applicatifs utilisent des bases de données, la plupart utilisant un modèle de données relationnel. Avec la complexité croissante des systèmes ainsi que les changements d'exigences émerge le besoin d'adapter et de faire évoluer les applications pour satisfaire à de nouveaux objectifs. Dans le contexte des applications utilisant des bases de données, les adaptations peuvent être effectuées au niveau de la base de données (par exemple, changement de schéma, migration de données) ou au niveau du logiciel applicatif (par exemple„ le code du programme). Les changements effectués à un niveau nécessitent souvent des changements à l'autre niveau afin de garder l'entièreté du système fonctionnel. La synchronisation de l'adaptation entre les différents niveaux est souvent appelée problème de co-évolution de schémas/programmes.

Les transformations bidirectionnelles (*bx*) sont une solution possible à ce problème de co-evolution entre la base de données et l'application. Elles peuvent être utilisées afin de découpler l'évolution du schéma de la base de données de celle du code du programme, par exemple, en autorisant des changements sur la base de données pendant que certains programmes peuvent rester inchangés. Dans ses travaux, J. Terwilliger introduit le concept théorique de *Channel* comme un mécanisme permettant de découpler les "bases de données virtuelles" utilisées par le code de l'application, de l'actuelle représentation des données maintenue au sein du système de gestion de base de données (ou encore "schéma physique").

Dans cette thèse, nous rapportons sur les considérations et les expériences d'implémentation de ces Channels, en pratique, dans le contexte d'une application complexe du monde réel. Nous nous focalisons sur les Channels implémentant les transformations telles que celles de Pivot et Unpivot, présentons différentes alternatives pour générer ces Channels et discutons les caractéristiques de performance de ces transformations à grande échelle. Nous présentons enfin un outil permettant la génération de ces Channels.

Acknowledgements

This work is the result of a three-month internship at the University of Victoria in Canada, British Columbia, where we were integrated in a research project aiming to build a *Primary Care Research Network* (PCRN) integrating several *Electronic Medical Record* (EMR) software systems.

The PCRN aims to connect the different EMR systems in order to share clinical information which will be used for medical research and data mining. We worked on one of these EMR systems called OSCAR, *Open Source Clinical Application Resource*, on which we contributed to a database migration process, aiming to improve maintainability and prepare the data for the integration in the PCRN.

Part of the three months of work resulted in the participation and formal presentation of our work at the *Bidirectional transformations (bx) – Theory and Applications Across Disciplines* workshop in Banff (December 1-6, 2013, Banff, Canada); and the publication of a paper for the *third international workshop on bidirectional transformations* (March 28, 2014, Athens, Greece) which was entitled : *Bidirectional Transformations in Database Evolution: A Case Study "At Scale"*.

Throughout this work, we received help from many people and we would like to thank all of them for the precious time and support they offered.

Foremost, we would like to express our sincere gratitude to our thesis promoter Anthony Cleve, for his support; for the time dedicated to late meetings during our internship; and for his patience, motivation, enthusiasm and expertise. His valuable advice helped us throughout the internship and while writing this thesis. We could not have imagined having a better advisor.

Besides our promoter, we would like to thank Jens Weber, our internship supervisor, for all the time he devoted to us; for the wonderful welcome in his research lab; and for making our stay in Victoria unforgettable. His encouragements, insightful comments and research questions helped us a lot while working in his team.

A particular thanks goes to Morgan Price for the support and opportunities he offered us and for the energy he transmits to the Symbioses lab team.

Our sincere thanks also go to Jeremy Ho, Anita Katahoire, Fieran Mason-Blakley

and Raymond Rusk, our Simbioses labmates, for their help and the good company during the months we spent in their office.

Last but not least, we would especially thank our families for the support they provided us through our entire life and in particular, we must acknowledge our respective partners, Marie and Justine, without whose encouragement and assistance we would probably never have finished this thesis.

Contents

List of Figures

Chapter 1

Introduction & Motivation

1.1 System evolution

Nowadays, information systems (IS) are everywhere and a major part of our daily life. As part of our world, these systems aim to capture real-world concepts that are potentially changing and as such, programs need to evolve to meet the requirements depending on this constantly evolving environment. According to Lehman [28], those programs must evolve because they "operate in or address problem or activity of the real world". Lehman refers to programs that mechanize a human or societal activity as *E-Programs*. Due to the nature of those programs, questions of correctness, satisfaction and appropriateness arise, leading to pressure for change and evolution.

There are various reasons for this need of evolution, in which we can mention the emergence of new technologies, the requirements changes, the lack of experts in certain technologies and so on.

System evolution has been studied for many years and today, it has been established that system conception and software engineering are evolving processes. The classical and well-known waterfall model was certainly the first attempt to structure the discipline of *software engineering*. This process can be decomposed in three main phases [29] :

- System Definition : Disciplined and structured analysis of the real needs and objectives to produce system requirements.
- Implementation : Implementation of the system specification. This phase includes the code validation, inspection and testing.

- Maintenance : Revision of the system to observe, report and correct faults. Minor adjustments to the software are supposed to happen here too.

Although this process has been formalized and accepted as a part of the IEEE 1219 Standard for Software Maintenance [24], it turned out to be too strict and rigid for supporting system evolution in an efficient way.

Today, this model is rarely used for developing systems liable to change. Instead, we have seen the emergence of new *evolutionary processes*, allowing to go back to earlier phases of the process in order to cope with continuous change.

Again according to Lehman [28], software evolution follows certain laws, such as continuing changes, increasing complexity, continuous growth or declining quality. Those laws enforce the need for information systems evolution.

It is now accepted that systems need to evolve and the maintenance phase is now extended with the evolution of the system. Software evolution is therefore an active and well-respected research topic in software engineering.

1.2 System comprehension and reverse-engineering

While evolving and maintaining information systems, it is crucial to have a deep and fine-grained understanding of the system since a poor understanding will often result in quality degradation and defaults, leading to an IS with different behaviours than the ones expected. When considering data-intensive IS evolution, it is important to note the different subsystems involved, such as the program itself and the database. Most of the time, the comprehension of these subsystems depends on the documentation available. Unfortunately, it is rare that such documentation exists and, when it does, it is nearly always obsolete (e.g. [25]). It is therefore important to recover this documentation for the system to be maintained and evolved in an effective way.

In some cases, program comprehension is realized through a reverse-engineering process. By analyzing the code, the user interface, the data-flow, the database, etc. it is possible to retrieve a good understanding of the information system. The reverse-engineering process is a complex and costly process but techniques and automated tools can help.

1.3 Database and program co-evolution

As mentioned in the previous section, information systems are made of multiple subsystems. The evolution of such a system implies therefore the evolution of all the underlying subsystems. Also, evolving only one subsystem often implies the evolution of the remaining ones.

Information systems often rely (nearly always) on DBMS for managing the large quantities of data they use and so, evolving the IS is, in most cases, followed by the evolution of the related database. Unfortunately, the mapping between the IS data view and the actual representation in the DBMS is not always straightforward. This conceptual gap is often referred to as the *Object-relational impedance mismatch* when using object-oriented languages.

Considering these arguments, IS evolution is not always simple and managing evolution that requires the synchronization of different layers is one of the challenges that computer science research is concerned about. It is also often referred to as the program/database co-evolution problem.

Although there is an abundant quantity of related works about this co-evolution problem, most of the theories/tools proposed cannot be applied directly. It is therefore important to analyze the IS context and choose between the appropriate potential solutions. Once the choice of using a given tool is made, there is still a requirement to implement a concrete solution or adapt an existing tool for the specific context of use.

1.4 Motivation and thesis statement

The specific information system on which we worked is called “OSCAR”. OSCAR is one of the commonly used EMR (Electronic Medical Record) systems in Canada. The aim of this system is to provide a complete solution for doctors to manage their everyday practices. As any system dealing with real-world concepts, OSCAR is also subject to pressure for change and evolution. There are several factors that push the system to evolve and we present here two of them that motivated our re-engineering work.

For technical reasons, OSCAR developers have decided to use the InnoDB engine

for all the database tables. Without re-engineering, some tables were unfortunately too large for using InnoDB and were still using MyIsam. Replacing these large tables with equivalent smaller structures is therefore a solution for migrating all the tables to the InnoDB engine.

EMRs are of a high interest in medical care improvement. By analyzing and allowing data mining on those information sources, it is possible to extract meaningful information to improve medical care in clinical practices and to provide more appropriate treatments to patients. The OSCAR EMR is an extremely rich source of data for data mining purposes but, unfortunately, some constructs of the database need specific programs to extract information. Through a process of re-engineering, we propose a new data structure that allows easier data integration and querying for data mining purposes.

The aim of this work is to positively reply to evolutionary needs by effectively applying the re-engineering process to the existing OSCAR system. As side effect, the evolution will also allow an easier understanding and maintenance of the database.

Considering that OSCAR manages highly critical data, the purpose of this thesis is to present the theoretical concepts for a *safe* information system evolution and to propose concrete implementations that are applicable in the context of a large and complex real-world application. By *safe* evolution, we mean that all the data will be preserved as well as their semantic. The behaviour of the system will also be kept intact.

1.5 Thesis limit

This thesis does not pretend to address all the problems of information system migration and evolution. As presented in Chapter 2, this problem has been under active research for many years and will continue to be studied even more in the future. However, we aim here to present the design of one possible implementation of a real-world solution for a critical information system migration which will help to maintain the system and simplify his continuous development and integration with other systems.

1.6 Thesis structure

Chapter 2 presents the state of the art related to information systems and their evolution. We present in this chapter different concepts that can help with performing *safe* migration and evolution. This chapter aims to give an overview of the previous works and existing approaches in these disciplines. Chapter 3 formalizes the design of our specific but generic solution for the migration. Chapter 4 shows the implementation of the previously designed solution. Chapter 5 exhibits the tool created that aims to help generating the code for the migration. Chapter 6 reports on a case study to validate the developed solution and verify its applicability for a large real-world application. It also presents a program-code refactoring in order to compare the performance with the solutions presented in Chapter 4. Chapter 8 concludes the thesis and opens additional discussions about future works aiming to improve our solution.

Chapter 2

State of the art

In this chapter, we present the state of the art related to information systems maintenance and evolution. First of all, we introduce the general concept of maintenance and the different scenarios that it can encompass. After, we explain the concept of coupled transformations in software evolution and present how encapsulation through wrappers can help in this situation. Then, we present and define the information systems (IS) and legacy information systems (LIS). Finally, we present some existing frameworks, theories and tools that can help while solving coupled-transformations scenarios, focusing on database evolution.

2.1 Software maintenance and evolution

Software maintenance is defined in the IEEE Standard for Software Maintenance as *the modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment* [24]. This definition states that software maintenance is a post-delivery activity but does not state what are the kinds of activities concerned.

Software maintenance is composed of various maintenance activities. A widely cited study by Lientz and Swanson [36] categorizes maintenance activities in four main classes :

- Adaptive: Maintenance activities to cope with changes in software environment
- Perfective: Maintenance activities to integrate new user requirements into existing software

- Corrective: Maintenance activities to fix errors in existing software
- Preventive: Maintenance activities to prevent problems in the future

From these four classes, the survey showed that more than 70% of the maintenance efforts were concentrated on the first two types. Many other subsequent studies show similar results and show that integrating new user requirements is the major problem in software maintenance and evolution.

Many authors also use the term of *software evolution* as a substitute for software maintenance. We consider here that software evolution is one post-delivery activity included in the process of software maintenance.

In one of their works, Bennet and Rajlich [5] announce that the conventional analysis of system maintenance and evolution from Lientz and Swanson is no longer useful as soon as modern systems are component-based, distributed systems, etc. In this work, Bennet and Rajlich therefore propose a stage-based model representing the software life-cycle as a sequence of different stages. The major contribution of this model is the separation of the "maintenance" phase into multiple stages, evolution being one of these. They also define the goal of software evolution as *adapting the application to ever-changing user requirements and operating environment.* This definition is similar to the one given by Lehman for *E-Programs* in [28].

2.2 Coupled software transformations

While considering migration and evolution of large software, many scenarios imply multiple artifacts. Lämmel [27] defines the *coupled transformations*, or *co-transformations*, as transformations scenarios involving two or more artifacts of potentially different types that are coupled in the sense that transformation at one end necessitates reconciling transformations at other ends such that global consistency is reestablished.

Coupled transformations are encountered in various disciplines of computer science and can be described formally as:

> *Let A and B be the types of the artifacts. We assume a consistency c relation on A and B. We are given two concrete artifacts $a : A$ and $b : B$*

such that $c(a,b)$ holds. We consider a type-preserving[1] *transformation on A, denoted by g, and we apply this transformation to a such that we obtain $a' = g(a)$. Then, the reconciliation issue is about determining a suitable b' such that $c(a',b')$ holds.*

Lämmel identifies four categories of coupled transformations ([27]) that we present below. In the following figures, the continuous arrows represent transformations and dashed arrows represent consistency relations.

1. **Coupled transformations : no reconciliation**
   The first category, *no reconciliation*, is depicted in Figure 2.1 and appears when g is known to be restricted such that a is changed without challenging the consistency. In this case, b can be kept as it is. This scenario appears for example when SQL manipulations are executed on the database without modifying the database schema.

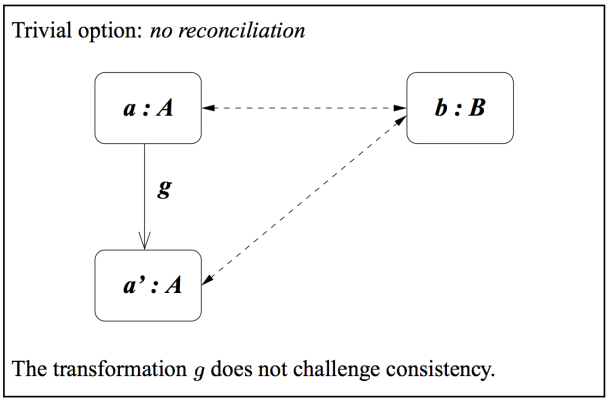


Figure 2.1 – Co-evolution: *no reconciliation* scenario (taken from [27])

2. **Coupled transformations : degenerated reconciliation**
   The second category, *degenerated reconciliation* is depicted in Figure 2.2 and appears when concrete artifacts of type B are derivable of artifact of type A, by the means of a translation t. For instance, code generated with domain-specific language (DSL) can be regenerated by the means of the translation t.

[1] A type-preserving transformations asserts the same type for input and output.

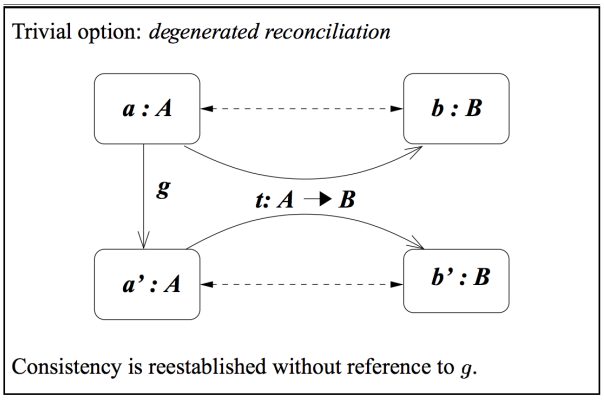


Figure 2.2 – Co-evolution: *degenerated reconciliation* scenario (taken from [27])

3. **Coupled transformations : symmetric reconciliation**
   The third category, *symmetric reconciliation*, is depicted in Figure 2.3 and is based on a transformation description f expressed in a transformation language. The interpretation of this transformation description f, denoted $\bar{f}$, provides two actual transformations: one on A and a second one on B. This scenario is typically encountered when reconciling a database instance in response to adaptations of a database schema.

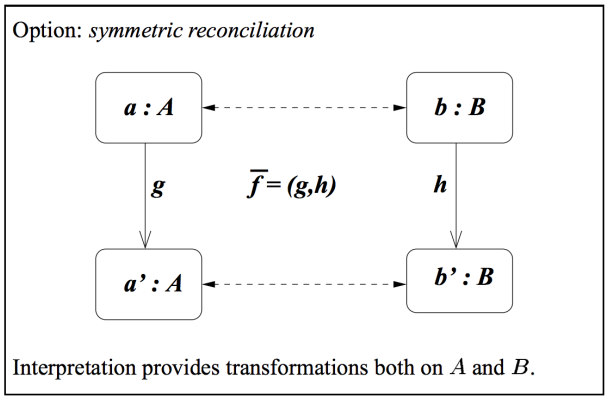


Figure 2.3 – Co-evolution: *symmetric reconciliation* scenario (taken from [27])

4. **Coupled transformations : asymmetric reconciliation**
   The last scenario, *asymmetric reconciliation* is more complex and out of scope for this work. We therefore redirect the reader to [27] for further information.

There are many scenarios that involve coupled transformations, including among others the view-update translation, co-evolution of design and implementation, etc. Although Lämmel proposes to categorize coupled-transformations into four classes,

he does not provide any solution for solving those coupled-transformation scenarios. In the next sections, we present the concepts of information systems and legacy information systems and then propose different ways of solving this co-evolution problem.

2.3 Information systems (IS)

The term *information system* (IS) is used by many authors and designates typically large-scale business application, comprising the software itself and its associated databases or other data management systems (DMS) ([21]). Usually, information systems are composed of many programs and data management systems.

Information systems are the backbone of the information flow for several companies and are often ageing applications running for decades. These systems are usually referred to as *legacy information systems* or LIS. As defined in [8], *a legacy information system is any information system that significantly resists modifications and changes.* Bennett [4] also defines a legacy software as a software which is vital to an organization, but which no one knows what to do with.

Some of the problems responsible for this situation are presented in [2] and [6], in which we can mention systems running on obsolete hardware, lack of documentation, poor program understanding or systems that are hard to evolve. It is also important to note that evolution will impact our current IS and that the current technology will become tomorrow's legacy.

2.4 LIS evolution and migration

According to Henrard & al. [23], legacy system migration is concerned with implementing a new system that preserves the functionality and data of the original system. In an ideal world, the migration of the data management system should not imply modifications in the software and the software should be developed regardless of any specific technologies. But, in practice, software is closely related to the data management systems and a modification on the former often implies modifications on the latter.

To achieve LIS migration process, many authors relate two main steps. Firstly, the database or the legacy data management technology has to be replaced by another one, potentially using another paradigm. Then, the application software must be adapted to use the target management system.

Related works ([21], [23]) assume that schema modification and data re-engineering can be considered as a DMS modification and that this process takes place in the overall process of IS/LIS evolution.

2.4.1 Wrappers for LIS evolution

While evolving legacy information systems, two main strategies are proposed [35]

- conversion
- encapsulation.

Conversion means that the code is rewritten or translated from the old environment to the new one, while *encapsulation* leaves the code in its current state and connects it through interfaces to the new presentation and access layer.

Software encapsulation is based on the "wrapping" technique. The term of *wrapper* in software evolution refers to the concept of wrapping legacy software components to integrate them in newer architecture. Figure 2.4 depicts the categories of wrappers identified by Orafli & al. [33]. A short explanation of the categories is given below.

- **Database wrappers:** Database wrappers are “gateways” to existing databases, allowing software to access legacy databases.
- **System services wrappers:** System services wrappers provide customized access to standard system services such as printing, etc.
- **Application wrappers:** Application wrappers encapsulate batch processes or online transactions, allowing client applications to use legacy components.
- **Function wrappers:** Function wrappers define an interface to invoke individual functions within a wrapped application, allowing the client application to call only certain parts of the legacy program.

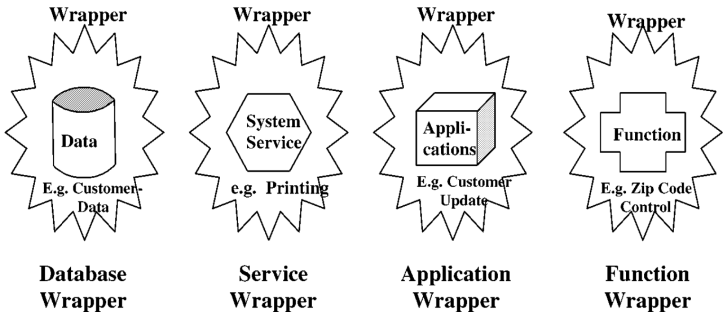


Figure 2.4 – Types of wrappers in LIS evolution (taken from [33])

2.4.2 A framework for software evolution

Considering the IS definition from Cleve and Hainaut [21], migration of IS raises two major issues. The first problem is the conversion of the database to a new DMS with its existing data. The second problem is the adaptation of the programs to the target DMS. In [45], Tilley and Smith present a high-level process of legacy information systems migration using 3 steps : (1) schema conversion, (2) data conversion and (3) program conversion.

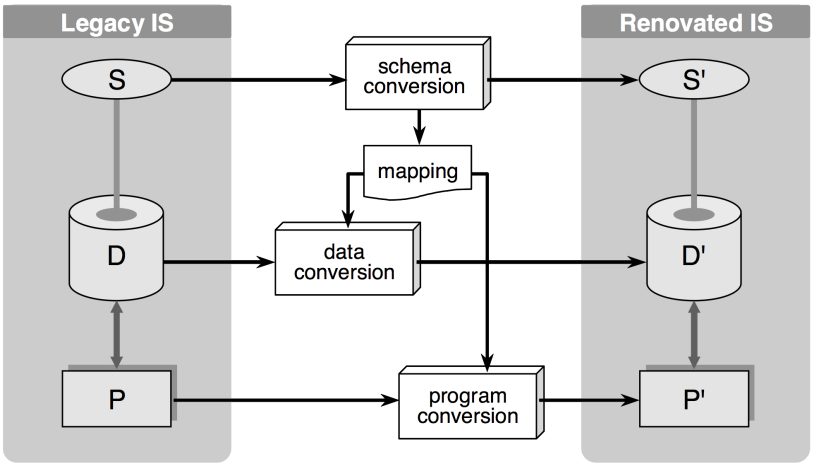


Figure 2.5 – Overall view of the *database-first* IS migration process (taken from [21])

Figure 2.5 depicts this process composed of the three steps, using a database-first migration scenario. The source database schema S is converted into a target database schema S' and a mapping between the legacy IS and the renovated IS is defined. Then, data is converted from the source schema to the destination schema, following previously defined the mapping rules. Finally, the source program P is converted

into the renovated program P'. The different steps are more precisely described below.

- **Schema conversion :** is the translation of the legacy database structure, or schema, into an equivalent database structure expressed in the new technology. Both schemas must convey the same semantics, i.e., all the source data should be losslessly stored into the target database. Most generally, the conversion of a source schema into a target schema is composed of two processes. The first one, called database reverse engineering, aims at recovering the conceptual schema that expresses the semantics of the source data structure. The second process is standard and consists in deriving the target physical schema from this conceptual specification. Both processes can be modeled by a chain of semantics-preserving schema transformations. Semantics-preserving schema transformations are transformations that, applied on a schema, produce a new schema in output with the same semantics. It neither increases nor reduces the existing semantics.

- **Data Conversion :** is the migration of the data instance from the legacy database to the new one. This migration involves data transformations that derive from the schema transformations described above.

- **Program conversion**, in the context of database migration, is the modification of the program so that it now accesses the migrated database instead of the legacy one. The functionality of the program is left unchanged, as well as its programming language and its user interface (they can migrate too, but this is another problem). Program conversion can be a complex process in that it relies on the rules used to transform the legacy schema into the target schema.

Other authors followed this way and proposed concrete evolution processes based on similar steps. The approach proposed by Cleve and Hainaut [21] is based on a model using two different axes : the data and the program dimensions. In this model they identify six possible strategies for evolving information systems. Their method is based on the *"database-first"* migration strategy that, as its name suggests, migrates the database first, allowing to build new applications on top of the new schema while incrementally migrating legacy programs.

The six global strategies identified in this framework are combinations of 2 strategies, respectively one for each dimension. A short explanation clearly inspired from [21]

of those strategies is given below, separated according to the data and program axes. The six strategies depicted in Figure 2.6.

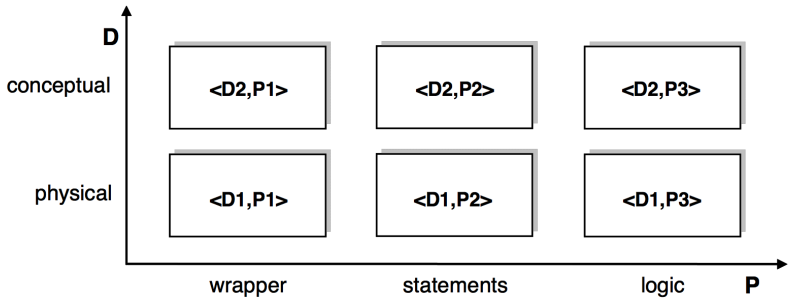


Figure 2.6 – The six reference IS migration strategies (taken from [21])

- **The database dimension (D)**: The authors consider two extreme database conversion strategies leading to different levels of quality of the transformed database.

  The first strategy (*Physical conversion* or D1) consists in translating each construct of the source database into the closest constructs of the target DMS without attempting any semantic interpretation. The aim of semantic interpretation of a database schema is to understand its underlying concepts, helping to translate source constructs into equivalent target constructs. The process is quite inexpensive, but it often leads to poor quality databases with no added value. In fact, misunderstanding in the original schema can lead to poor translation. On the other hand, if the quality of the source schema is good and if the schema is complete, this conversion can result in a precise conversion.

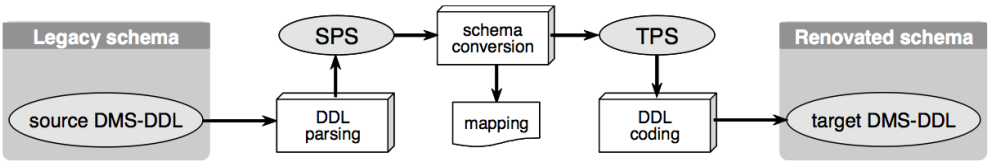


Figure 2.7 – Physical schema conversion strategy (taken from [21])

Figure 2.7 depicts this physical conversion. In the first place, the source DMS-DDL schema is extracted from the DBMS and translated into the source physical schema (SPS). This SPS is then converted into a target physical schema

(TPS) according to the mapping definition for the schema conversion. Finally, the DMS-DDL code is generated from the TPS.

The second strategy (*Conceptual conversion* or D2) consists in recovering the precise semantic description (i.e., its conceptual schema) of the source database first, through reverse engineering techniques, then in developing the target database from this schema through a standard database methodology. The target database is of high quality according to the expressiveness of the new DMS model and is fully documented, but, as expected, the process is more expensive. For more information on database schema re-engineering through conceptual conversion, the reader can refer to Hainaut [20].

Figure 2.8 depicts this conceptual conversion. In the first place, the physical schema is extracted into the SPS. Then the SPS is refined by integrating other sources of information as the program source code or by analyzing existing data. This refined schema is then used to create the source logical schema (SLS) before being conceptualized in a conceptual schema(CS)(using the Entity-relationship model for example). Those steps output in a mapping definition. The CS is then translated in the target logical schema (TLS) and then in the target physical schema according to the mapping rules defined before. Finally, the DMS-DDL code is generated in order to create the renovated schema.

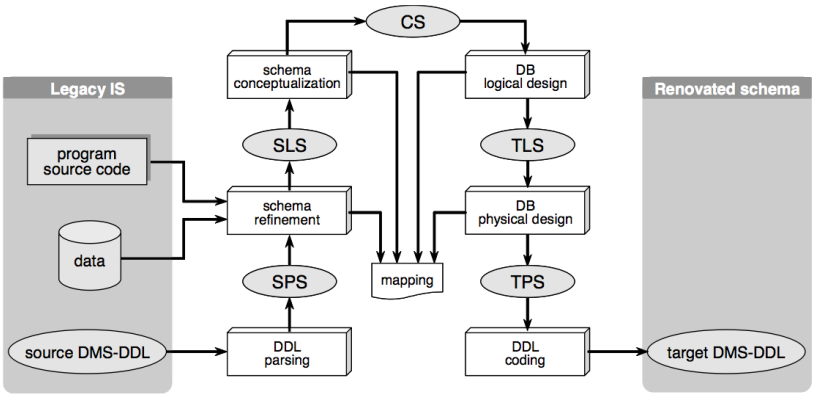


Figure 2.8 – Conceptual schema conversion strategy (taken from [21])

- **The program dimension (P)**: Once the database has been converted, several approaches to application programs adaptation can be followed. The

authors identify three reference strategies for program migration.

The first one (*Wrappers* or P1) relies on wrappers which encapsulate the new database to provide the application program(s) with the legacy data access logic. Figure 2.9 depicts the database migration using a wrapper-based approach where the wrapper provides an interface similar to the legacy database access logic by simulating the source physical schema (SPS) on the top of the new target schema (TPS).

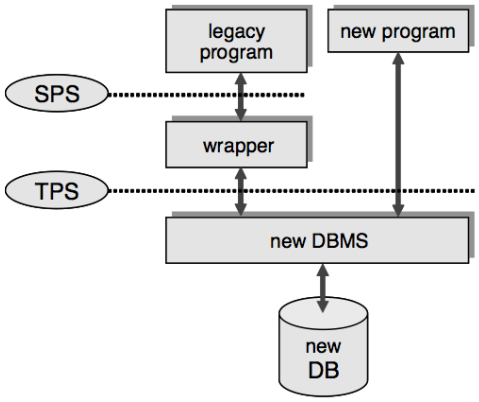


Figure 2.9 – Wrapper-based migration architecture (taken from [21])

The second strategy (*Statement rewriting* or P2) consists in rewriting the access statements in order to make them process the new data through the new DMS.

According to the third strategy (*Logic rewriting* or P3), the program is rewritten in order to use the new DMS-DML at its full power. It requires a deep understanding of the program logic, since the latter will generally be changed due to, for instance, the change in database paradigm.

2.5 Database wrappers for LIS migration

Legacy information systems evolution often implies the integration of legacy components in new architectures. In the specific case of schema evolution, the database physical model and the program data view can be quite different. A

common strategy used to reconcile these different models is the use of *wrappers*.

In [43], Thiran & al. mainly refer to two main categories of wrappers : the forward wrappers (*f-wrappers*) and the backward wrappers (*b-wrappers).* The forward wrappers, corresponding to the *"database-last"* migration strategy [2], leave the database in its original state and new programs are built using modern practices. A wrapper then takes care of translating queries to the legacy database. The backward wrappers, corresponding to the *"database-first"* migration strategy ([2]), deal with migrating the legacy database first. Legacy programs are adapted (if needed) to use a wrapper that simulates the legacy DMS. This strategy allows new programs to be built on top of the new DMS, exploiting all its benefits. Figure 2.10 depicts the two wrapper categories explained before.

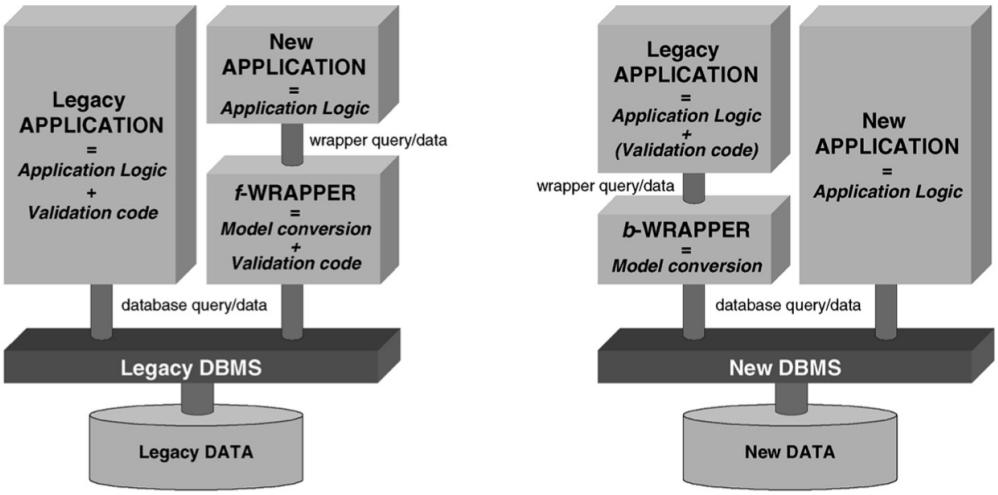


Figure 2.10 – Wrappers categories (taken from [43])

In other words, wrappers are intermediate layers that "simulate" another behaviour of a data source through a generic interface. Those wrappers can be defined using schema transformations. A schema transformation is defined by Thiran & al. [44] as follow : *"A schema transformation consists in deriving a target schema S' from a source S by replacing construct C (possibly empty) in S with a new construct C' (possibly empty). A transformation T can be completely defined by a couple of mappings <T,t> where T is called the structural mapping and t the instance mapping : $C'=T(C)$ and $c' = t(c)$. T explains how to replace construct C with construct C' while t states how to compute instance c' of C' from any instance c of C."*

Those transformations can be compound, leading to complex transformations. They allow to define mapping between the source and the target schema, helping to define and generate the wrappers.

Another important aspect of wrappers that Thiran & al. [44] address are semantics-preserving wrappers. So called wrappers have an invertibility property such that a transformation T on a schema C can be inverted with a transformation T' so that $C = T'(T(C))$ and an the instance transformation t on c instance of C can be inverted with a instance transformation t' so that $c = t'(t(c))$.

Many previous works also relate to automatic wrapper generation techniques. [42] and [44] present processes for automatic wrapper generation but always in the context of wrapping legacy databases to simulate newer technologies. Few works concern wrapping migrated databases to interface with legacy programs, a.k.a *backward wrappers*. Henrard & Al. provide a complete overview of this discipline ([22]).

2.6 The view-update problem

Relational views can be interpreted as concrete implementations of wrappers in the database domain. Unfortunately, the problem of updating source data with a wrapper is similar to the *the view-update problem.*

Many applications need to have a specific view of the data. In LIS migration, for instance, legacy programs have to be interfaced with new DMS through backward wrappers because they need the old data source view. This scenario is a concrete example that is largely used in industry.

According to Dayal, Ramamritham and Vijayaraman ([12]), *for a view to be useful, users must be able to apply retrieval and update operations to it.* Then, the operation on the views must be translated to the corresponding operations on the source DMS.

In [3] , Bancilhon and Spyratos present the foundations of the updatable-views theory . They describe the difficulty of this problem as : *"the definition mapping is sufficient to translate view queries into the database queries. View updates, however, present a difficult problem : A database update that translates a view update must take the database to a state mapping onto the updated view. Now there is, in general, more than one database update that satisfies this requirements. The*

problem is how to choose one ...".

This description can be formalized as the following : *"given a target model T specified as a set of views over a source model S by a set of queries Q, determine if it is possible to intercept updates to T and instead, update S such as that re-running queries Q on S regenerate the updated instance of T exactly. The update to S must be unambiguous, i.e., there can be exactly one way to do it." [12].*

Another work of interest is the one of Melnik & al. [32] in which they explain how it is possible to compile the mapping between two schemas into bidirectional views. The concept of bidirectionality is explained below. In this work, Melnik & al. also give an interesting note that complex operations as joins and unions on views are usually not, or not easily, updatable.

2.7 Concepts of bidirectionality

Bidirectional transformations (*bx*) are a mechanism for maintaining the consistency of two (or more) related sources of information. Researches in many areas have been undertaken to investigate how the use of *bx* could potentially solve many computer sciences related problems, including the relational view update, schema evolution, data exchange, database migration and so on. Although there has been a little cross discipline interaction and cooperation until now, *bx* sub-communities emerge more and more and *bx* are today used in a variety of fields including programming languages, database management systems, model-driven engineering and other disciplines ([11]).

The basic idea of bidirectional transformation can be expressed as follows : *"there exist two models or schemas S and T (sometimes referred to a source and target model), and a mapping between them M. The mapping serves as a bridge to allow operations and data to flow between the two models, and must conform to bidirectional properties that govern the quality of the synchronization between S and T . How the mapping M operates, how it is specified, and what services it offers consumers of the target schema T differ greatly depending on what technology is used to build M. However, regardless of how M is built, it must ensure that any consumers of either S or T are able to access their data with no surprises."*[40]

From this definition, we can understand that bidirectional concepts are applicable

in both ways. Furthermore, still according to [11], *bx* between two sources of information A and B comprise a pair of unidirectional transformations. Therefore, we usually call *forward transformations* the transformations from A to B, and *backward transformations* the ones from B to A. Although this idea of a pair of unidirectional transformation is quite effective and interesting, today's researches have proposed a new approach where *bx* are based on bidirectional transformation languages. A major advantage of those languages is that the forward and backward transformations are defined simultaneously and the bidirectional property of transformations can be guaranteed by construction.

One of the major concepts of bidirectional transformations lays in a pair of functions : the *get* and the *put* functions. The *get* function allows to extract a view T from some complex data structures S and the *put* function allows to put back the updates on this view T into the original structure S. This concept has been formalized in [18] as the concept of *lenses* and serves for many of today's available *bx* tools and theories.

Over time, lenses have become more specialized and multiple categories of lenses appeared. What was originally called *lenses* is now referred to as *classical lenses* or *asymmetric lenses* to avoid ambiguity with other concepts.

The original idea of lenses is comparable to the idea of an updatable view. The function *get* allows to extract a view and the function *put* defines the update policy in order to update the data source.

Functions *get* and *put* have, in the sense of a classical lens or asymmetric lens, the following definition [39].

$$get : S \to T \qquad\qquad put : T \times S \to S$$

In a few words, the *get* function transforms a store state S into a target state T without any context required. For the other function however, a context is required. The *put* function uses the original store state S as the context when translating an updated target state T to a new store state S.

Lenses following this definition are interesting but most of the formalism around lenses is based on a subset that is *total* and *well-behaved*.

In short, *total* lenses are lenses whose pair of functions are both total on their inputs and *well-behaved* lenses are lenses where the *get* function can map all the arguments from the data source S to the target view T and the *put* function can map back all the argument from the view T to the data source S.

To ensure that a lens is *well-behaved*, it has to satisfy two round tripping policies. In [39], Terwilliger & Al. write: *First, there is the intuition that, given a target state t, if one pushes that state to the store s and retrieves it again, one always gets the original state as a result. Using the original formulation of a classical lens, this property — also called "PutGet" — amounts to:.*

$$\forall_{(t \in T)} \forall_{(s \in S)} get(put(t, s)) = t$$

Second, there is the intuition that, given a store state s, if one applies a lens to it and immediately pushes the result back, one gets back the original store state. This property — also called "GetPut" — amounts to:

$$\forall_{(t \in T)} \forall_{(s \in S)} put(get(s, s)) = s$$

Now that the basic concepts of lenses and bidirectionality have been presented, the rest of this section will present the major techniques available today in the domain of *bx*, focusing on databases.

2.8 Bidirectionality in databases

Bidirectional transformations are widely used in the database domain. They represent a good way to find solutions for schema evolution and to solve the view update problem. By developing and making use of tools that allow automatic or semi-automatic software evolution and support query rewriting, it is possible to save time and money on the software development process.

Over time, *bx* tools have supported different operations and satisfy different formal properties. We therefore propose a comparison of some of them in the next sections. In order to process to this comparison, we have to define which features and capabilities such tools may exhibit. This comparison is inspired by [40].

- **Information capacity** considers whether the tool can construct a mapping that alters the information capacity of the models on which it operates. A mapping between a source model S and a target model T can, depending on the case, decrease the information capacity, increase the information capacity or preserve the information capacity.

- **Concrete versus Virtual state** considers if the target model states will be materialized or not. For instance, one can create a concrete view and propagate the changes to the underlying original source model and another can define a dynamic view.
- **Support for model evolution** considers how the tool copes with evolution in either of its models.
- **Support for coupled evolution** considers how the tool is able to support coupled evolution, as presented in Section 2.2.
- **Support for complex transformations** considers if the tool can handle complex transformations. An example of complex transformations can be the Pivot and the Unpivot operations. Pivoting is a transformation that takes data in separate rows, aggregates them and converts them into columns. Unpivot is its dual function.

In the next section, we present three tools for supporting *bx* in databases and then provide a comparison of those tools by comparing some of their characteristics.

2.8.1 Relational Lenses

Recent studies in bidirectional database transformations have been undertaken over the development of smaller algebraic transformations with known properties. Relational Lenses Framework [7] proposes a novel approach to the view update problem. In relational lenses, Bohannon & al. aim to define a bidirectional query language, in which every expression can be read both as a view definition (the *get* function) and as an update policy (the *put* function), with a set of primitives and type system specifically targeted to relational data, including relational databases. The primitives of the language they define is based on relational operators.

In relational lenses, Bohannon & al. provide a detailed analysis of the view update behaviour of a number of fundamental relational operations. They mainly focus on three operations : the join, the selection and the projection. They aim to provide bidirectional versions for those relational operators, by providing definitions of the *get* and *put* functions.

Thanks to the restriction to *total* and *well-behaved* lenses, the relational lenses framework allows to compose simple transformations to create complex ones,

preserving their *bx* properties. Unfortunately, even by composing transformations, some relational database operations cannot be described in the relational lenses framework.

The relational lenses framework is based on the lenses framework and show nearly the same properties. Firstly, it supports decreasing information capacity. Symmetric lenses also support increasing information capacity, through the means of a complement. Lenses are based on materialized target states in the sense that when the user update the target model state, the *put* function takes care of the correct translation of the update into a new source state. It does not support model evolution as it, neither co-evolution nor complex transformations.

2.8.2 PRISM

PRISM is an integrated solution that provides support for database schema evolution, including, among others, schema transformation, data migration and query rewriting.

PRISM is based on the most recent results on mapping composition, mapping invertibility and query rewriting. These major concepts are integrated in the theory of Schema Modification Operators (SMOs), which serve as basis for PRISM. [10] defines an SMO as *a function that receives as input a relational schema and the underlying database, and produces as output a (modified) version of the input schema and a migrated version of the database.*

SMOs have been designed to provide a simple operational language for database administrators to express the evolution of relational schemas through different versions with integrity constraints. Used in combination with integrity constraints, SMOs are guaranteed to be invertible. Of course, SMOs are combinable and a composition of SMOs is, by nature, invertible as each operator can be studied in isolation. Combined with the logical framework of Disjunctive Embedded Dependencies (DEDs), SMOs allow to define the forward and backward functions to support schema evolution. DEDs extend the classical embedded dependencies with disjunctive and non-equality dependencies so that they can be used to express all the common relational integrity constraints such as foreign keys and referential integrity ([13]). Although DEDs in combination with SMOs are effective, some inverses are still not expressible. For specific cases, PRISM relies on user interaction to select an inverse among various candidates.

The reader can refer to [17] and [16] for further information on the invertibility properties of operators.

PRISM is based on a 5-step process. We shortly present the steps and redirect the reader to [10] for a detailed explanation.

1. **Evolution Design :** In this step, the DBA expresses by the means of SMOs, a schema modification. The system translates the SMO into a logical mapping between the schema versions and then a DED-based chase engine is exploited to rewrite queries from the original schema version to the target schema version.

2. **Inverse Generation :** After the schema modification, the system computes the candidate inverse sequences, based on the original SMOs sequence and integrity constraints. Possible multiple inverses are disambiguate with the use of integrity constraints or user interaction. After a check on information preserving properties of SMO inverses, the schema evolution is guaranteed, or not, to be completely reversible.

3. **Validation and query support :** The inverse SMOs sequence is then translated into a logical mapping between the new schema version and the old schema version and queries on the new schema are translated into equivalent queries on the original schema.

4. **Materialization and Performance :** Once the mapping between the schema versions is defined and queries can be expressed in an equivalent way on each schema version, the system automatically translates the forward SMOs sequence into a SQL data migration script in order to migrate all the instances from the original schema to the new schema.

5. **Deployment :** In this last step, the original tables are dropped and recreated through the means of SQL views.

The SMO language has its advantages and its drawbacks. Its advantages include the fact that its syntax is close to SQL making it easy to learn. Also, it can help the DBA in their day to day tasks tasks by providing automatic support for schema versions management, query rewriting and automatic data migration. Unfortunately, some SMOs are quasi-inverse, which means that in some cases, the inverse transformations may lose information.

SMOs sequences, such as the ones used in PRISM, show increasing and decreasing information capacity. PRISM operates on a virtual target state but can also support a materialized target state, especially for data migration. Unfortunately, it does not handle the co-evolution. Although it provides an interesting tool, PRISM's operator set is limited to the major relational operators. It does not provide support for complex transformations as the ones discussed in this work.

2.8.3 Channels

In the Channels works ([41], [38]), Terwilliger & al. relate to the concept of *virtual database* for the understanding of the data schema at the program level. Although the term "virtual database" can refer to the concept of classical view, it expresses more than that, including schema modification support, extended update support, etc. The concept of *Channels* is part of an overall work known under the name of *Guava (GUI As View) Framework*.

Channels are presented as an alternative means for defining a virtual database and its mapping to a physical database that guarantees they remain synchronized under data and schema updates against the virtual schema [41]. It is important to see that, in this simple definition of what a *channel* is, the authors address not only the classical view update problem (that is concern about data) but also how it is possible to propagate DDL schema modifications on views to the underlying source model.

The theory presented in this work is based on the concept of channel transformations (CTs). Channels are a composition of atomic transformations that are used to define mappings between the virtual database and the source database, with known and provable bidirectional properties. Of course, composing atomic transformations with known properties allows one to build more powerful transformations, still having those properties. Where other tools are unable, Channels allow to express some complex transformations such as Pivot and Unpivot, which are of high interest for our work.

Channels support decreasing information capacity for some operators that are not fully bidirectional. They do not support increasing information capacity. Channels are a *bx* solution that allows to support *true* virtual databases in the sense of virtual state database. In comparison to relational lenses, that use a state-based approach to resolve the view update problem, Channels translate DML and DDL statements

directly. Each CT that receives queries expressed in relational algebra on the virtual database produces in output queries expressed in extended relational algebra on the native schema. Classical update, insert and delete DML statements are supported by Channels. The reader can refer to [41] for the complete set of update statements that is supported by channels. Channels also support model co-evolution.

Although Channels seem to be perfectly suitable for database schema evolution and query rewriting, they only provide support for a subset of CTs in Entity Framework, the Microsoft proprietary object-relational mapping tool ([41]).

2.8.4 *bx*-tools comparison

We present in Table 2.1 a short summary comparison of the features and capabilities of each tool.

	Inf. cap.	Concr/Virt	Model Evo.	Co-Evo.	Complex transfo
Relational Lenses	+ , -	Virt.	NO	NO	NO
PRISM	+ , -	Both	YES	/	NO
Channels	-	Virtual	YES	YES	YES

Table 2.1 – *bx* tools comparison

For the specific purpose of this work, we want to use a virtual state-based tool able to support some complex transformations such as those presented in Chapter 3. Considering those arguments, Channels were the most suitable choice for our specific purpose.

Chapter 3

Design

This chapter presents the conceptual design of our solution. As this solution is driven by specific needs (although it can be applied in various contexts), we introduce here some basic concepts in order to facilitate understanding. The reader can refer to chapter 6 for a detailed explanation of the OSCAR system and the case study.

OSCAR is the information system under consideration for this work. In short, OSCAR([30]) is an Electronic Medical Record system (EMR) used by practitioners to facilitate the management of the patients and their data. The particular software evolution problem faced by the OSCAR LIS lies in re-engineering a large number of large, sparse tables in order to increase maintainability of the system and to allow the OSCAR users to migrate to a newer database storage engine, which cannot handle such large tables.
The chapter is structured as follows: Section 3.1 presents the data structure pattern used for the data management system evolution, Section 3.2 presents the concrete design of a specific and type-safe structure responding to the specific needs of this application context and Section 3.3 presents the transformation definitions that will be used to define the mapping between the versions of the database schema structures.

3.1 The Entity-Attribute-Value model

3.1.1 EAV model presentation

When storing sparse data into a database, the Entity-Attribute-Value model (a.k.a the EAV model) is often considered. The EAV model uses a set of tables for storing

data in a generic way. It is also sometimes called “vertical database”. Although it is described as an anti-pattern (B. Karwin[26]), the EAV model proved to be highly efficient for specific purposes such as the storage of sparse medical records ([9], [14], [15], [34]).

In fact, one of the main benefits of the EAV model lies in its generic and extensible structure capacity. The relational model, as good as it can be, has problems when coping with changes to the data structure. Most of the time, a change in the data structure implies a change in the related database table and as a result, any change in the application schema leads, most of the time, to a refactoring of the database structure. The EAV model does not face the same issue and seems perfect for system evolutions without database schema modification. [26] defines the EAV model as it :

> *"The solution that appeals to some programmers when they need to support variable attributes is to create a second table, storing attributes as rows."*
> *Each row in this attribute table has three columns:*
>
> - ***The Entity :*** *Typically this is a foreign key to a parent table that has one row per entity.*
> - ***The Attribute :*** *This is simply the name of a column in a conventional table, but in this new design, we have to identify the attribute on each given row.*
> - ***The Value :*** *Each entity has a value for each of its attributes.*
>
> *This design is called Entity-Attribute-Value, or EAV for short. It is also sometimes called open schema, schema-less, or name-value pairs.*

3.1.2 EAV model limitations

The EAV model is perfectly suitable for storing large size data structures containing highly sparse data, like those often used for some specific medical purposes. However, using it implies sacrificing many advantages offered by the conventional relational database model. For example, it is impossible to define mandatory attributes and it is not easy to use SQL data types, enforce referential integrity, etc. But although the model does not allow these kind of constraints, it is still

possible to manage them on the program side.

The OSCAR EMR manipulates medical forms which are of a specific data structure. These forms store various medical data such as examinations or blood test results. Due to the general nature of these forms, they contain a large number of fields, only few of which are filled per patient.

For this specific work, some constraints had to be enforced, and providing a type-safe solution for the EAV model was of high importance. As we work on a medical information system, this EAV model aims to store critical information and ensuring that meta-data (data type, default values,...) are preserved is part of ensuring the data quality. It is also important to preserve the original data type in order to provide a solution respecting the original types by, for example, providing a view with correct data types.

3.1.3 The OSCAR's Forms custom EAV model

In the next sections, we present the designed solution for the migration of the forms related tables to an EAV model, the decisions that were taken, and how problems have been solved. For the sake of clarity, all the examples presented in the chapter use a simple EAV model without data type support.

The following simple example depicts the problem that had to be solved. Figure 3.1 presents a small form, containing some fields and check-boxes. This form stores its data in a relational database using a conventional normalized relational schema, just as it is working for the forms and the related tables in the database.

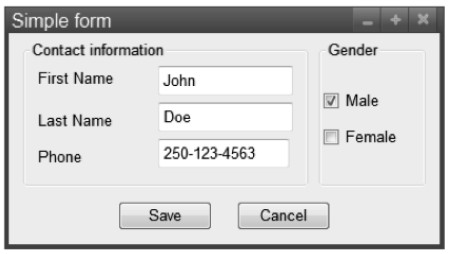


Figure 3.1 – A simple form example

A concrete instance of the associated table in the schema could be Table 3.1.

ID	FirstName	LastName	Phone	Gender
1	John	Doe	205-123-4563	M
2	...	...	...	...

Table 3.1 – Conventional relational table

The number of inputs in the source form defines the number of columns in the destination relational table. As a result, a complex form containing a lot of fields will always result in a related relational table containing a lot of columns. It is exactly here where the EAV model makes sense. The aim of the EAV model is to store the same information but in a one-row-per-attribute format, using a limited set of columns.

The operation of transforming a relational table into an EAV model is comparable to rotating the records and create one row per attribute for each row in the original table. Moving a table from 10 columns containing 10 rows will result in an EAV table made of 10 rows * 10 columns = 100 rows in the resulting EAV table. This operation is often referred as the "Unpivot" operation.

An EAV model example, storing the same information that the conventional relational table presented in Table 3.1 could be Table 3.2.

Entity	Attribute	Value
1	FirstName	John
1	LastName	Doe
1	Phone	205-123-4563
1	Gender	M
2	FirstName	...
2	...	...

Table 3.2 – EAV table

This data storage structure, although it has some limitations, shows some great benefits especially for medical data storage. In fact, the model resolves the problem of data sparseness by storing only attributes with non-null values in the EAV table. So in many cases, a table composed of 10 columns having 10 rows will often generate less than 100 rows in the EAV table, depending on the original table sparseness .

Although this data structure is highly efficient for solving the data sparseness problem, it has however some limitations. Presenting the limitations of this model is out of scope for this work and we therefore mainly focus on two open questions

that will define the remaining part of our work. The reader can refer to [26] for more information on the EAV model.

The two main challenges implied by the use of an EAV model can be summarized by the following questions.

1. *How is it possible to ensure that all the data will be preserved?*
    - *Are the transformations loss-less?*
    - *Are the transformations semantics-preserving?*
2. *What are the reversibility properties of this transformation?*
    - *Is it possible to continue to use the legacy system with the new database structure?*
    - *Is it possible to wrap the evolution to ease the system migration?*

From those questions arise two main properties: *provability* and *reversibility*.

In order to provide a solution having those properties, we focused first on how to preserve all the data and meta-information from the original schema (SQL constraints, data type, data length, data themselves, etc.). We then worked on how to prove all the transformations steps from the original tables to the generic EAV schema and allow to have provable reverse transformations in order to define views on the EAV model.

3.2 A generic and type-safe EAV model

Designing an EAV model for our specific needs is an important step of the work. The *type-safety* and *reversibility* properties were the two main challenges when designing our EAV model. It is important to give them a special attention in order to achieve the goal of having a generic, loss-less and semantic-preserving model.

Unfortunately, the EAV table, in its simplest expression, does not make possible to use data types, as soon as all the attribute values are stored in the same table column. Research on supporting data types with the EAV model has already been conducted ([14],[34]). Three different approaches for supporting types in the EAV model are proposed in these works.

1. **The multi data type EAV schema:** This schema uses multiple tables, one for each data type.

2. **The hybrid EAV schema:** This schema uses multiple columns in the EAV table, one column for each data type.

3. **The Variant data type:** This schema uses a variant data type to store the different data type. This solution has performance limitations and may not be offered in many DBMS systems. (It is not offered in MySQL, for example, the DBMS used by OSCAR.)

Considering the different possibilities above, the first and naive proposed solution was to store all the attribute values in a unique column and keep the meta-data of the original table in another table in order that it would still be possible to retrieve the original data type with the EAV model. All the attribute's values were stored with a common data type (VARCHAR(255)) and the only way to enforce the data type safeness was to check on the program side if the data to be inserted respected the original data type. This solution uses the same idea than using a "variant" data type and also had the same benefits and issues.

Having only one column for attribute value storage simplifies the use of the model (especially when selecting values) and also makes it easier to maintain but, as a drawback, some casting operations are needed in order to preserve the data types. In addition, this technique makes use of an extra disk space usage for some data type representation.

3.2.1 Description of the naive EAV model

The first implementation presented in 3.2 is composed of 4 tables.

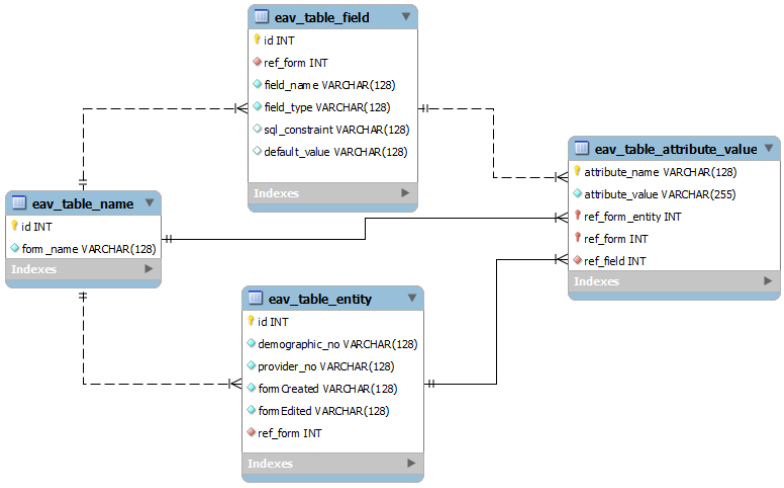


Figure 3.2 – Naive Entity-Attribute-Value model

1. **Eav_form_attribute_value :** Is the main table of the model that is the EAV table. The table contains the entity id (ref_form_entity field), the attribute name (attribute_name field) and the attribute value (attribute_value field). The table only stores non-null attribute values. The attribute value is stored using a VARCHAR(255) data type.

2. **Eav_form_entity :** This table stores the entity fields that are kept in a conventional representation. It depicts the concrete entity with the non-sparse attribute subset. We use this table to store the common attributes between all the forms tables.

3. **Eav_form_field :** This table stores the field information for each form table that has been migrated to the EAV model.

4. **Eav_form_name :** This table stores the name and id of the form, allowing to create a link between the form name and an ID used in the other tables.

This simple model suffers from several limitations. First, not all the data types fit into a VARCHAR(255) field and the same problem arises when using a TEXT field.

Then, condition-based queries will be less efficient on a VARCHAR2 field since only VARCHAR2 comparison is possible. For instance, ordering on number is no

more possible when $12 < 2$ in a ascending order on a VARCHAR2. Considering the existence of casting functions, the reader could think that it is not really a problem. We however argue that some casting operations do not exist, the casting functions exist only for a subset of data types.

Finally, it is impossible to query the table and obtain the original data type in the result set. All the data types will be of the attribute value column type and the types have to be managed on the application side. Once again, although a possible solution of casting types on the database side seems possible, the specific DBMS used in OSCAR (MySQL) does not provide casting functions for some data types starting from a VARCHAR field.

The "BLOB" data type faces exactly the same problem of casting on the database side. It is not possible to cast a blob field in anything else on the database side.

3.2.2 Description of the extended EAV model

The first designed EAV model fails to meet data type preservation and its use necessitates a lot of casts in order to use it. Due to the limitations of the first naive EAV model implementation, a new version has been developed. For this implementation, the EAV model uses multiple value columns, one for each data type. Using multiple tables has also been considered, but, as soon as the OSCAR developers team wanted to reduce the number of table in the schema, it was not useful to use an EAV schema needing 10 or more tables.

The choice we made for the OSCAR case is to use a hybrid EAV schema with one table and multiple columns types. Since this may result in a potentially large number of columns, our transformation implementation generates an EAV schema to consider only those data type that are really needed in the original tables.

Figure 3.3 presents a concrete example of this EAV schema. The surrounding tables are still present and store meta-data on the original tables and fields. The EAV table will store a value only in the column corresponding to the original data type, leaving other columns with a null value. This model now stores all data, with the original data type.

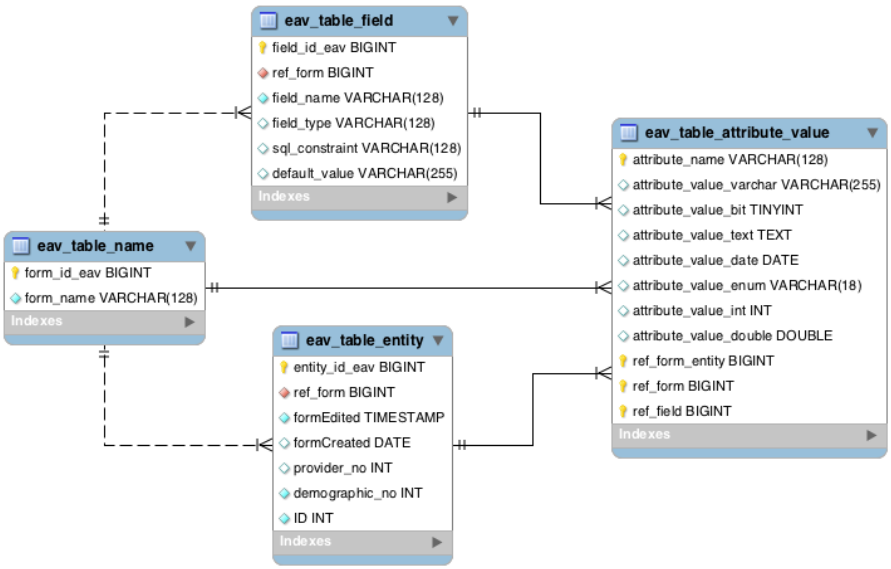


Figure 3.3 – Final Entity-Attribute-Value model

The next steps are to define a loss-less transformation, in both ways, for data migration and for view reconstruction to allow the legacy system to make use of this new data structure. In order to avoid large amount of code refactoring (complicated by the fact that there is no documentation on the system), we decided to use a wrapper-based technique to recreate the original view of the data for the programs. In Section 3.3, we present the definitions of the transformations used to define the data migration and to recreate the original data view from the EAV table.

3.3 Transformation definition

In this section, we provide definitions for the primitive and composite transformations used in this work. Those definitions are either identical theoretical definitions or adaptations from J. Terwilliger [37].

The issue of type-preservation also implies that it is impossible to use the "Pivot" and "Unpivot" functions that are sometimes defined in certain DBMS. As soon as we have to manage multiple columns in the input for the pivot function or in the output for the unpivot function, we have to define our own implementation.

Another detractor of using Pivot/Unpivot operators provided by some DBMS is that they are not well-defined and lack a unified semantics ([46]). It is therefore not possible to predict what will be the output in specific cases as for example, a

non-unique id for the pivot function.

Several relational algebra operators are used in the following sections. Although people belonging to the database community certainly know those operators, we shortly present the ones we use here for people unfamiliar with relational algebra.

- Projection (π) : The projection of a relation onto a subset of its attributes (table columns) is a relation formed by restricting some of the columns. We list the attributes we want to project in the result as a subscript to π.
- Selection (σ) : The selection is a relation onto a subset of itself containing the rows respecting a condition. The conditions are listed as a subscript to σ.
- Rename (ρ) : Applying the rename operator to a relation produces the identical relation except that the relation and its attributes are given new names. The new names are specified as a subscript to ρ. $\rho_{V \to C1}$ means that the original attribute V will be renamed in C1 in the renamed relation.
- Natural Join ($\bowtie$) : The natural join is used to combine relations that have some attributes in common, based on the equality of commons attributes.
- Left Outer Join ($⟕$) : The left outer join is like the natural join but replacing the data that would be lost with nulls as values where there is no corresponding values in the set of the right side of the operator.
- Right Outer Join ($⟖$) : The right outer join is like the left outer join but replacing values from the right side of the operators with null values.
- Delta (δ) : Eliminates duplicates from bags. Even if relational algebra works with bags, we use some extended algebra operators to depict some important concepts in order to facilitate the implementation.

Readers can refer to [19] for more information on relational algebra and its related operators.

3.3.1 Pivot and Unpivot transformations

The *Pivot* operator (T'=PIVOT(T,A,V)) transforms a table T in generic key-attribute-value form into a form with one column per attribute. Column A must participate in the primary key of T and provide the names for the new columns in

T', populated with data from column V. The resulting table is named T'. The formal definition given for the Pivot operator using relational algebra is presented below and based on the work of J. Terwilliger [37].

$$\nearrow\!\!\!\square_{\mathbf{C};A;V}\, T \equiv$$

$$(\pi_{columns(T)-\{A,V\}}T) \bowtie (\rho_{V\to C1}\pi_{columns(T)-(A)}\sigma_{A=C1}T)$$

$$\bowtie ... \bowtie (\rho_{V\to Cn}\pi_{columns(T)-\{A\}}\sigma_{A=Cn}T)$$

$$for\ \ C1,...,Cn = \mathbf{C} = \delta(\pi_A(T))$$

Figure 3.4 shows the intermediate steps of the Pivot operation for an example table T with three columns. In this case, *Period* is the pivoting attribute A whose values will give rise to columns in the resulting table and *Price* provides the values for these columns. Figure 3.4 shows that intermediate relations are created for each arising attribute. The key for the resulting table T' will be all remaining columns in T (all columns other than A and V).

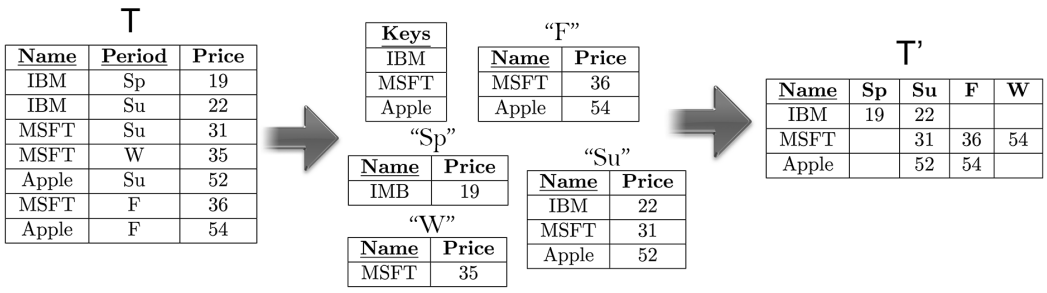


Figure 3.4 – T'=PIVOT (T,Period,Price)

The *Unpivot* operator (T'=UNPIVOT(T,A,V)) is the inverse of the Pivot operator and transforms a table T from a one-column-per-attribute form into key-attribute-value triples, effectively moving column names into data values in new column A (which is added to the key) with corresponding data values placed in column V. The resulting table is named T'. The formal definition (given by J. Terwilliger [37]) for this operator in relational algebra is presented below.

$$\not\boxtimes_{\mathbf{C};A;V} T \equiv$$

$$\bigcup_{c \in \mathbf{C}} (\rho_{C \to V} \pi_{columns(T)-(\mathbf{C}-\{C\})} \sigma_{C<>null}(T)$$

$$\times \rho_{1 \to A}(name(C)))$$

Figure 3.5 shows the intermediate steps of the Unpivot operation for an example table T. First, the original table T is broken down into intermediate relations that correspond to the set of upivoted columns. The original column names are used as values for the "Period" attribute and the actual value of the column is used to define the "Price" attribute values. Then, the intermediate relations are merged into an unique relation.

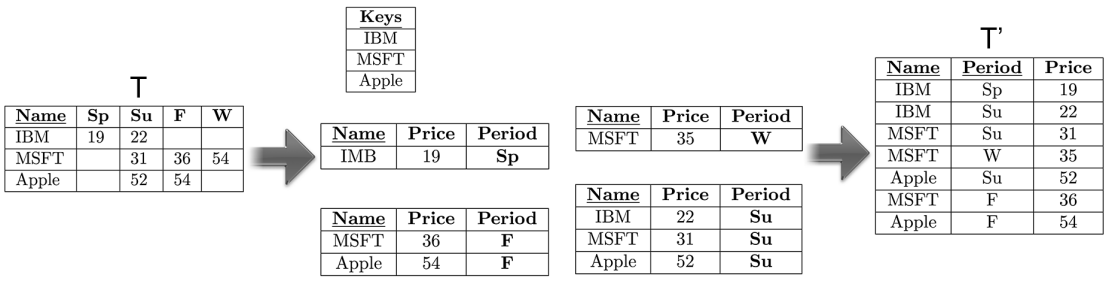
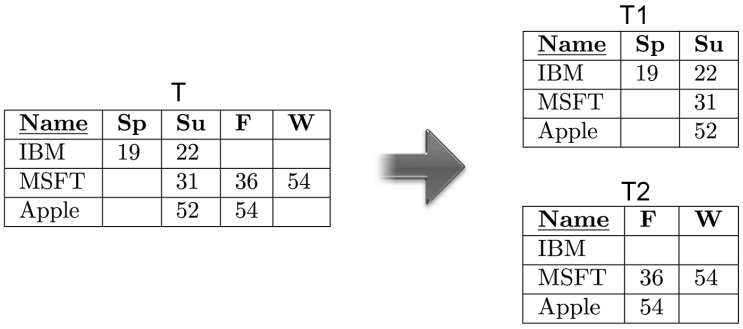


Figure 3.5 – T'=UNPIVOT(T, Period, Price)

3.3.2 VPartition and VMerge transformations

In practice, Pivot and Unpivot transformations are often used in composition with two other operators, referred to as *VPartition* and *VMerge* in [37].

The $(T1, T2) = VPartition(T, f)$ operator splits a given table into two tables $T1$ *and* $T2$, according to a total selection function f, which associates each non-key column with one of the two target tables ($T1$ or $T2$). Both resulting tables share the key columns of T. Figure 3.6 shows the intermediate steps of the VPartition operation for a short example.



T

Name	Sp	Su	F	W
IBM	19	22		
MSFT		31	36	54
Apple		52	54	

T1

Name	Sp	Su
IBM	19	22
MSFT		31
Apple		52

T2

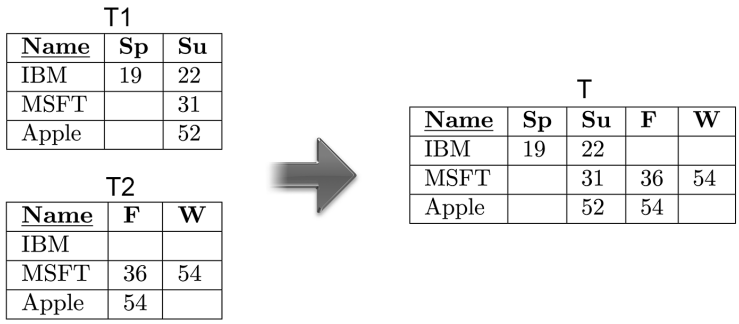
Name	F	W
IBM		
MSFT	36	54
Apple	54	

Figure 3.6 – (T1,T2)=VPartition(T, f) with f(x) = true iff x ∈ (Sp, Su)

The formal definition of this VPartition can be defined as :

$$VPartition_{\mathbf{C}}T \equiv (\pi_{\{T-\{f(t)\}+A\}},\ \pi_{\{f(t)\}})$$

The $(T = VMerge(T1,T2))$ operator is the inverse of the $VPartition$ operator and reconstructs a single table T using two tables $T1$ and $T2$ sharing a common attributes set C. Figure 3.7 presents a simple example of the VMerge transformation.



T1

Name	Sp	Su
IBM	19	22
MSFT		31
Apple		52

T2

Name	F	W
IBM		
MSFT	36	54
Apple	54	

T

Name	Sp	Su	F	W
IBM	19	22		
MSFT		31	36	54
Apple		52	54	

Figure 3.7 – T=VPartition(T1, T2)

The formal definition of this VMerge can be defined as :

$$VMerge_{\mathbf{C}}(T1,T2) \equiv T1 \bowtie_C T2$$

3.3.3 Complex transformations: *create/get/put*

Complex transformations (and the Channels that implement them) are composed of the concatenation of primitive transformations, such as the ones defined above. The

composite transformation we will focus on in our case study combines VPartition and Unpivot to transform database structures in one-column-per-attribute format into equivalent structures into an Entity-Attribute-Value (one-row-per-attribute) format (akin to *create* and *put* in the lens framework ([7])). The inverse transformation composes Pivot and VMerge to reconstruct the original structure (akin to *get*).

The *create* function in this case will be implemented by an algorithm to migrate data from the original schema to the EAV model. The *put* function is typically the *wrapper* that transforms queries against the "original" table (reconstructed by a view here) to equivalent queries against the EAV schema. The *get* function is the original table reconstruction through a view definition.

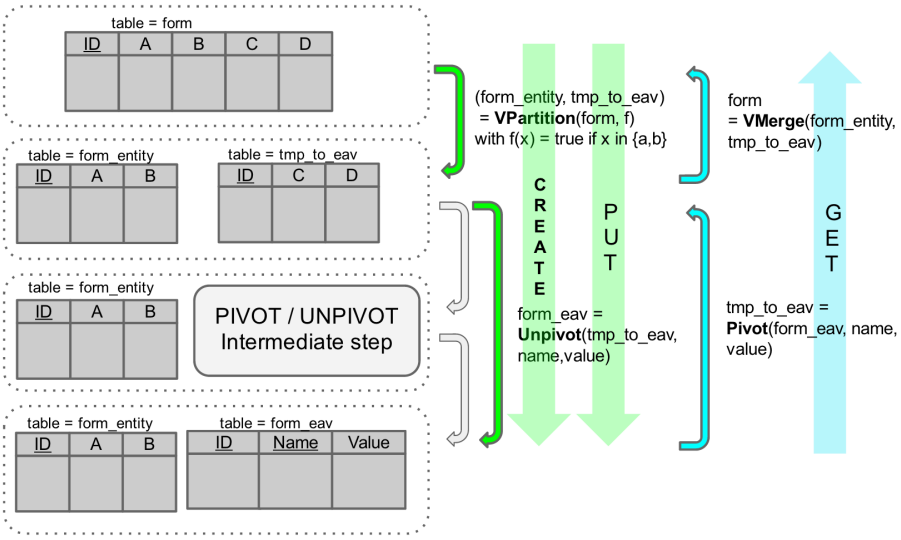


Figure 3.8 – Composite BX - *create/get/put*

Figure 3.8 illustrates these transformations with a graphical example. In this Figure, the source table (form) is first partitioned in 2 tables using the VPartition operation. Then, the Unpivot operation transforms the table tmp_to_eav into an target EAV table (form_eav). The *create* function depicts the initial data import in the EAV model. The *put* function allows to put back the update in the source table to the new target model structure. On the other hand, the *get* function allows to define a view by executing the inverse transformations. First, the eav_table is transformed into a classical one-column-per-attribute table using the Pivot operator. Then, the result of this operation is merged with the table form_entity to recreate the source table structure.

3.3.4 The view-update problem with the EAV model

As mentioned above, the *put* function is a backward function, allowing to query our EAV model through translated queries on the EAV model. There are 4 main categories of operations in the relational database model, often called the "CRUD" operations, respectively for Creating, Reading, Updating and Deleting row(s).

Here, we want to emphasize the fact that this function does more than only transforming queries from the EAV model to the original table through the view mechanisms. It is also important to allow the view to be adaptable and so, it means that the view should propagate changes (Create, Update and Delete operations) to the underlying database structures.

This view update problem could seem easy to solve but it is not and in most cases, represents a lot of work. Further explanation about a theoretical way to solve this problem is presented in chapter 4. Another practical way applicable in this context is also presented in Chapter 7 where we present a solution combining the use of Channels and code-refactoring.

Figure 3.9 illustrates by a simple example how the view is constructed and how it propagates the changes to the original data sources. The sum of blue (uninterrupted) arrows represents the *get* function, the purple (interrupted) ones represents the *put* function. The transformations executed are listed in the small rounded rectangles and the overall picture shows how the update is propagated to the original data sources through the different steps.

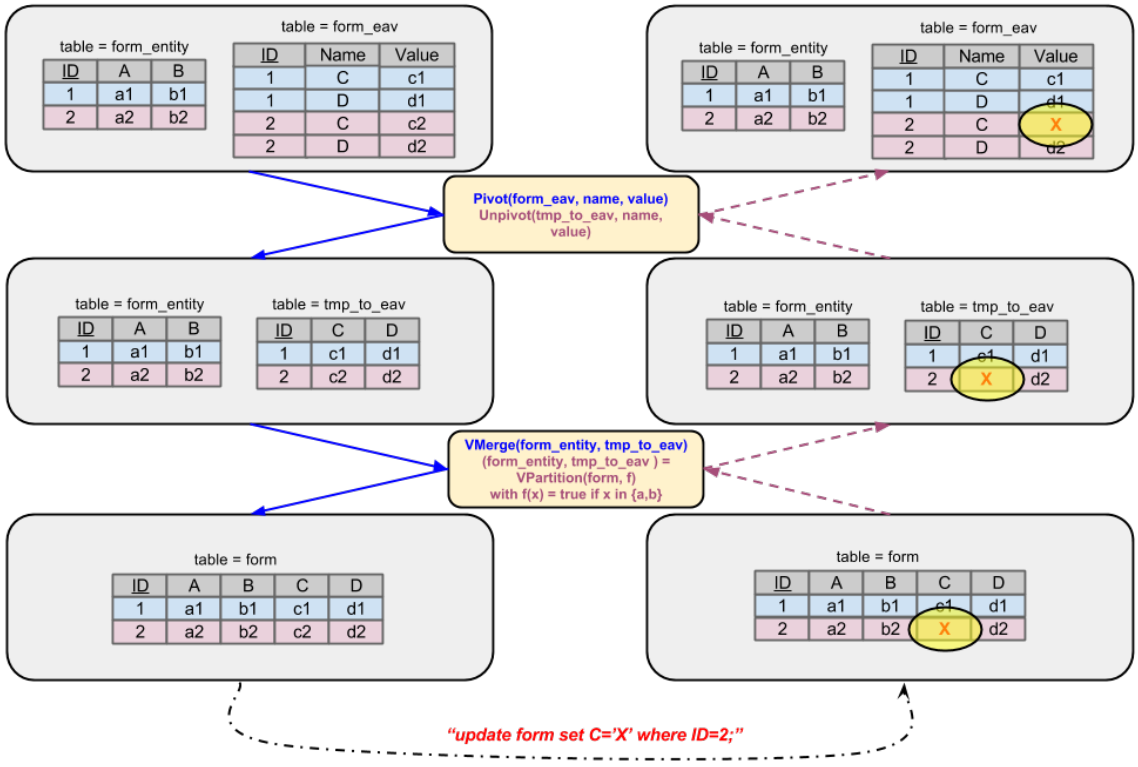


Figure 3.9 – View update problem : Concrete illustration

Chapter 4

Implementation

This chapter presents a concrete implementation of the functions previously introduced in Chapter 3. For each table that was migrated to the EAV model, the functions (*get*, *put*, *create*) have to be implemented. In order to help creating those functions, a code generation tool has been developed to generate the implementation code. The description of the functions implementation and the tool support have been separated into two chapters. The concrete implementation is presented here while Chapter 5 gives details about the tool implementation.

The "Get" and the "Put" functions are well-known in the relational lenses framework and in this chapter, the functions, the framework and those concepts of bidirectionality will be linked to the specific case of OSCAR. In the relational lenses framework, authors also define the function "*create*". This function can be seen as the initialization of the system, the initial import of the data into our new EAV model. We provide here the definition for this function too.

Different strategies and techniques can be applied when implementing Channels for the previously presented transformations. This section describes and compares such alternatives and presents a new technique referred to as the "coalescing approach". In this chapter, we assume that the database management system (DBMS) used does not have built-in operators for Pivot and Unpivot transformations. Indeed, although they are available in some DBMS, most of them still lack these operators. Moreover, even if they are present, their semantics is not standardized ([46]).

In almost all examples from the sections below, some (SQL) pseudo-code will be given in order to help the reader understand the implementation. All the examples are based on the tables from the schema presented in Figure 3.8 and use a simple

EAV model. We decided not to consider type-safety information in the examples in order to keep them easily understandable. The type-safety property, although it is important, does not add big challenges in the implementation. The main difficulty was to define which column was used for specific field depending on its type.

In the next sections, we present the implementation of the *create* function in Section 4.1, the *put* function in Section 4.2 and the *get* function in Section 4.3.

4.1 Implementation of the "Create" function

In our application domain, namely database evolution, the *create* function is mainly used for the data migration task, i.e., to transform data that conform to the "old" schema into equivalent data conforming to the new schema structure (a.k.a. the EAV model). The amount of data may be large in real-life applications. This step can be implemented with a database client program (e.g. Java applications) or directly within the database server via a stored procedure. We implemented both alternatives and present below the different performance results obtained in terms of performance.

In the next sections, we provide two possible implementations for the *create* function. First, we present a procedural approach using a MySQL procedural language and then we propose another method to migrate the data with a single query on the DBMS.

4.1.1 The procedural approach

The procedural approach can be implemented on the database side as well as on the client side. The first alternative (database client program) is much less efficient than the second one. The client side program transfers the data from the database to the client program and then re-inserts the transformed data into the new schema. However, it has the benefit of being more platform independent by abstracting from the manipulation a specific DBMS procedural language (as PL/SQL in Oracle,Transact-SQL in Microsoft SQL Server). By parametrizing the JDBC (Java DataBase Connectivity) connector in the java application, we could potentially address all the DBMS vendor types in only one application.

Given the efficiency of the database side procedural approach, we therefore only present the implementation of this solution. However, the abstract algorithms are similar for both approaches and a high-level algorithm is presented as Algorithm 1.

Algorithm: Data migration : Create function

Data: T a set of tables and C a set of columns for each table

```
for t ∈ T do
    INSERT the table name IN the eav_table_name table;
    INSERT fields information from set C IN the eav_table_field table;
    for row ∈ T do
        INSERT the data kept in a one-column-per-attribute (A) form IN the
        eav_table_entity table;
        for attribute ∈ row\{A} do
            if attribute value is not null then
                INSERT the remaining attributes in the
                eav_table_attribute_value table;
            end
        end
    end
end
```

Algorithm 1: Data migration algorithm

The procedural approach uses nested loops. For each table, the first loop inserts each entity in the “entity table" (the table containing the columns that will not be unpivoted, e.g., the entity keys and any columns that should remain in the original format) and, for each inserted entity, a second loop is executed in order to insert the unpivoted attributes in the corresponding Entity-Attribute-Value (EAV) table.

In the concrete implementation, using the procedural language for MySQL, it is impossible to loop on columns of a specific record. MySQL only allows one to loop on rows, not on columns. Therefore, we materialized the loop on the columns through a large set of IF statements.

An example using the tables from Figure 3.8 is presented in Figure 4.1. Although it is a simple example, it helps the reader to project this solution for such big tables as they can exist in real software systems. First, the procedure loops on all the rows from the original table. For each row, it inserts the columns kept in the original format into the entity table and then, it checks for each attribute if its

value is non-null. If not, it inserts the attribute in the EAV table for the given entity.

```
BEGIN
DECLARE id_var,A_var,B_var,C_var,D_var INTEGER;
DECLARE cur1 CURSOR FOR SELECT * FROM form;
OPEN cur1;
 read_loop: LOOP
 FETCH cur1 INTO id_var,A_var,B_var,C_var,D_var;
 IF (no more records){LEAVE read_loop;}
 INSERT INTO eav_table_entity VALUES(id_var,A_var,B_var);
 ...
 IF(C_var is not null or C_var !=""){
 INSERT INTO eav_table_attribute_value VALUES(id_var,"C",C_var);}
 ... (for all the unpivoted attributes)
 END LOOP;
CLOSE cur1;
END
```

Figure 4.1 – Procedural approach (Create)

4.1.2 The declarative approach

Although the procedural approach works effectively, we decided to investigate another way of implementing the *create* function in order to solve some performance issues. We decided to use another technique and to avoid loops present in Algorithm 1. Algorithm 2 presents the idea of inserting nearly everything in the EAV model in only one INSERT statement. This implementation uses unions of selects (one select for each destination column) that recreate the EAV table directly and just need to be inserted in the corresponding table. This algorithm is nothing more than a translation of the Unpivot operation definition given in Figure 3.5 from relational algebra into SQL.

Algorithm: Data migration : Create function

Data: T a set of tables

for $t \in T$ **do**

  INSERT the table name IN the eav_table_name table.;
  INSERT fields information (type, constraint,...) IN the eav_table_field table.;
  INSERT the data kept in a one-column-per-attribute (A) form IN the eav_table_entity table.;
  INSERT **all** the remaining attributes IN the eav_table_entity_value table in one single statement.;

end

Algorithm 2: Data migration algorithm

Similarly to the declarative approach, the algorithm starts by inserting the table name and the fields information in the corresponding tables. It also inserts the entity into the appropriate table and instead of inserting all the attributes one by one (loop), it executes only one insert query composed of one union of selects to migrate all the unpivoted attributes into the corresponding table. This approach allowed us to cut down the time needed by one order of magnitude of 10 when migrating the data from the original schema to the EAV model. However, it needs to configure MySQL to manipulate a larger amount of files at the same time.[1]

The SQL pseudo-code is given in Figure 4.2. First, the columns kept in the original format are inserted in the entity table an then the remaining columns are inserted into in the EAV table through the union of selects.

[1](–open-files-limit=xxxxxx)

```
BEGIN
 INSERT INTO eav_table_entity SELECT id,A,B FROM form;
 INSERT INTO eav_table_attribute_value SELECT * from (
 SELECT id,name,value FROM (SELECT id,C AS value FROM form WHERE C IS NOT NULL),(
     SELECT "C" AS name FROM DUAL)
 UNION
 SELECT id,name,value FROM (SELECT id,D AS value FROM form WHERE D IS NOT NULL),(
     SELECT "D" AS name FROM DUAL))
 ... (Union ... )
END
```

Figure 4.2 – Declarative approach (Create)

4.2 Implementation of the "Put" function

The put function is used to maintain the actual database synchronized with the virtual database. The virtual database, a.k.a. the original tables recreated through the views, is not directly impacted by CRUD (Create, Read, Update and Delete) operations. In fact, every time the software creates, removes or updates a form, the related data have to be propagated to the underlying EAV model.

As we decided to construct our virtual database based on common DBMS mechanisms such as views and triggers, we had to deal with the limitations of the DBMS functions set. The easiest and the most transparent way to catch a query executed on a view and perform the corresponding insert into the EAV schema is to use triggers. Triggers have theoretically the property to fit both state-based (Relational Lenses) and query translation (Channels) approaches. In this case we use "Instead of" trigger that translate the query fitting therefore the Channel approach. As a first step, we created them manually in order to produce a proof-of-concept and to evaluate the feasibility of this method. Then, we developped a generic approach to generate those triggers automatically.

4.2.1 The insert

The insert is managed by an *instead of* trigger that executes pre-defined code instead of the original insert query. Every time the user wants to insert a new row into the virtual database, the trigger extracts the data from the query and propagates inserts into the EAV schema, managing the structural changes. This is performed in two

steps:

- Firstly, it inserts the attributes kept in the original format into the entity table and retrieves the auto-generated id for each entity.

- Then, it inserts every other attributes (unpivoted attributes) into the EAV table, using the entity id generated before in order to make the link between the attributes (and the attributes' values) and the entity.

Regarding the forms structure, the identifiers can be composite. For this migration to a generic EAV model, we cannot rely on the uniqueness of this identifier and so, we decided to generate a technical identifier for the EAV model. However, in order to keep the overall system working efficiently, we had to keep the original identifier in a horizontal format.

The manipulation is presented in Figure 4.3, accompanied by pseudo-code in Algorithm 3 and the SQL code of the trigger in Figure 4.4. All the examples of this section are based on Figure 3.8.

Figure 4.3 shows how an insert into the original table is translated to the EAV model. Some columns are kept in the original format and inserted into the entity table while the remaining columns are pivoted and inserted into the EAV table.

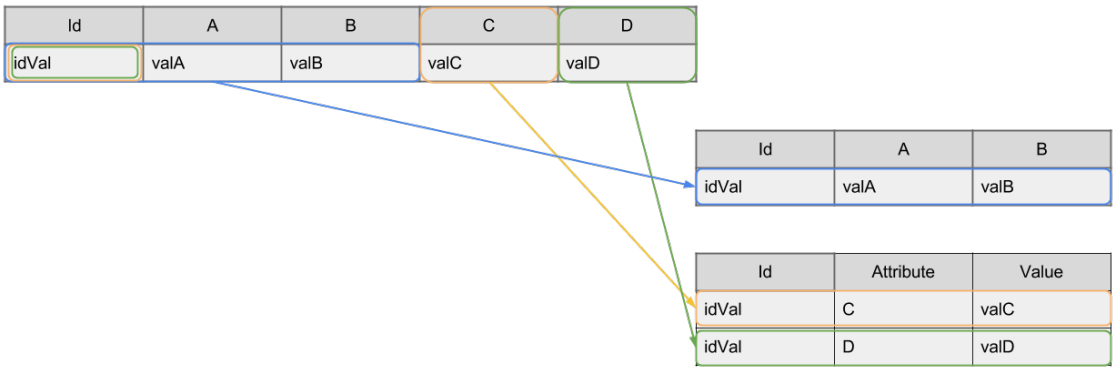


Figure 4.3 – Data insertion example

This manipulation is presented in Algorithm 3 which repeats the operation for each row inserted.

Algorithm: CRUD : Insert Trigger

Data: I a set inserted rows
for $row \in I$ **do**
    INSERT the data kept in a one-column-per-attribute form IN the eav_table_entity table.;
    for $attribute \in row \backslash \{A\}$ **do**
        if *attribute value is not null* **then**
            INSERT the remaining attributes IN the eav_table_entity_value table;
        end
    end
end

Algorithm 3: Insert trigger algorithm

In order to avoid data spareness, and therefore to optimize space usage, the null or empty fields are not stored in the EAV table. In the case of OSCAR's form, we make no difference between NULL and an empty value. In some systems, null and empty value may be semantically different, but here, in the existing DAL(Data Access Layer) implementation, every null value was inserted in the database as a empty string.

To convert this algorithm into SQL, we faced some difficulties. First, in the insert trigger, it was not possible to iterate on the column name. Therefore, we decided to flatten the loop, involving a considerable increase in the size of the trigger's code. It has to be noted that it does not impact the complexity of the algorithm. An example of pseudo SQL code for the insert trigger is presented in 4.4.

```
% SQL Code for Insert trigger
CRETE TRIGGER insert_form INSTEAD OF INSERT ON form_view
FOR EACH ROW BEGIN
INSERT INTO eav_table_entity VALUES(NEW.id,NEW.A,NEW.B);
IF(NEW.C IS NOT NULL){
INSERT INTO eav_table_attribute_value VALUES(NEW.id,"C",NEW.C);}
IF(NEW.D IS NOT NULL){
INSERT INTO eav_table_attribute_value VALUES(NEW.id,"D",NEW.D);}
END
```

Figure 4.4 – Insert trigger (Put)

4.2.2 The delete

Towards insert statement in SQL, the delete function is performed by a trigger that is executed on the EAV model. The query is simply translated into two delete statements on the EAV model. The first one deletes all the rows that refer to the selected entity in the EAV table, the second delete statement removes the entity from the entity table. Algorithm 4 presents the delete operation.

Algorithm: CRUD : delete Trigger

Data: I a set deleted rows
for $row \in D$ **do**
    DELETE the row that are linked to the form entity in eav_table_attribute_value.;
    DELETE the entity row from eav_table_entity.;
end

Algorithm 4: Delete trigger algorithm

This pseudo-code can be translated into corresponding SQL code without any specific adaptation.

```
%  SQL Code for Delete trigger
DROP TRIGGER IF EXISTS delete_form;
CREATE TRIGGER delete_form INSTEAD OF DELETE ON form_view
FOR EACH ROW
BEGIN
 DELETE FROM eav_table_attribute_value WHERE id=OLD.id;
 DELETE FROM eav_table_entity WHERE id=OLD.id;
END
```

Figure 4.5 – Delete trigger (Put)

4.2.3 The update

Finally, the update statement, which unsurprisingly is executed through a trigger too, propagates any update of a record to the EAV schema. In order to deal with the various behaviours that an update statement can induce, this trigger contains more code and is also more complex.

First, the trigger updates the row and field’s values that have been kept in the original format in the entity table as the original update query would do. Then, it

iterates on all attributes that are stored in the EAV format and manages the updates of each attribute independently. For each row, there are four different behaviours depending on the update effects:

- **Attribute value did not change :** in this case the trigger does nothing.
- **Attribute value changed from null (or empty) to a non null value :** the trigger then inserts a new row into the EAV table corresponding to the attribute name and its value.
- **Attribute value changed from non-null to null (or empty):** the trigger then deletes the row containing this attribute.
- **Update non-null value with another non-null value:** represents the case when the row exists but has a different attribute value. The trigger simply updates the row in the EAV table in order to change its value.

Algorithm: CRUD : update Trigger

Data: I a set updated rows
for $row \in U$ **do**
    Update the data kept in a one-column-per-attribute form in the eav_table_entity table.;
    for $attribute \in row \backslash \{A\}$ **do**
        if *OLD.column not like NEWċolumn* **then**
            if *OLD.column is empty* **then**
                The value was null before so we insert a new line containing the attribute and its value.;
            else
                if *NEW.column is null* **then**
                    Delete the row that contains the attribute.;
                else
                    Update the row that contains the attribute.;
                end
            end
        end
    end
end

Algorithm 5: Update trigger algorithm

For this trigger, like for the one managing the insert statement, it is not possible to iterate on the attribute values and to use the attribute name to select the

corresponding column. Therefore, we decided to flatten this loop too. As a result, the trigger code may be long (up to thousand of lines for large tables). Figure 4.6 shows the pseudo code of the update trigger for the previous example. For each updated row, the procedure updates the entity table. Then, it compares the value of the EAV attributes with the values stored in the EAV table. If the values are not present, it adds the attribute values to the EAV table, in other cases, it updates (or deletes) the existing values.

```
% SQL Code for Update trigger
CREATE TRIGGER update_form INSTEAD OF UPDATE ON form_view
FOR EACH ROW BEGIN
UPDATE eav_table_entity SET B=NEW.B,A=NEW.A where id=NEW.id;

IF(NEW.C not like OLD.C){
 IF(OLD.C==""){
  INSERT INTO eav_table_attibute_value VALUES(NEW.ID,"C",NEW.C);
 }ELSEIF(new.id is null){
  DELETE FROM eav_table_attibute_value WHERE name="C" AND ID=NEW.ID;
  ELSE
  UPDATE eav_table_attibute_value SET value=NEW.C WHERE name="C" AND ID=NEW.id;
 }
}

IF(NEW.D not like OLD.D){
 IF(OLD.D==""){
  INSERT INTO eav_table_attibute_value VALUES(NEW.ID,"D",NEW.D);
 }ELSEIF(new.id is null){
  DELETE FROM eav_table_attibute_value WHERE name="D" AND ID=NEW.ID;
  ELSE
  UPDATE eav_table_attibute_value SET value=NEW.D WHERE name="D" AND ID=NEW.id;
 }
}

END
```

Figure 4.6 – Update trigger (Put)

4.3 Implementation of the “Get” function

The get function is a major element of the virtual database. This function is used to create the view that will allow the legacy program to keep functioning without re-factoring the application code. To implement the get function, we first followed the formal definition presented in Chapter 3. However, we faced scalability problems with this solution that encouraged us to develop and present a second implementation.

4.3.1 The join approach

Our first implementation, which is inspired from J. Terwilliger work, is based on the Join operation. This query can be decomposed in two steps.

- First we select separately all the attributes into different sub-tables with as many "select as" as attributes. This step creates many two column virtual tables containing the id of the entity ("entity id") and the attribute for each migrated entity. Theses table contain as many rows as migrated entities (the null value is set in case there is no row for the current entity and attribute in the EAV table).

- The second step joins all those tables based on the "entity id" attributes in order to create the view.

Although this solution is quite easy to understand and to implement, it also has its drawbacks. First of all, the over-sized SQL query does not help code understanding and debugging. In addition, this query is typically slow. The execution time can reach minutes, depending on the number of rows in the EAV model. (More information about performance is provided in Chapter 6). Then, MySQL has a maximum-join-per-query limit of 61 tables. Depending on the number of form attributes, this technique cannot always be applied. For some forms, it was impossible to build back a view that has more than 61 attributes. A pseudo-code sample is available in Figure 4.7. The manipulation performed by the *get* function has already been presented in Section 3.3.3.
The following pseudo code illustrates how the view recreation works in practice. Left outer joins are executed in order to keep the single lines (add null when there is no corresponding row) and the select is executed on the overall join, using "select as" operation to define the original column names as column names for the view.

```
SELECT a.id,a.A,a.B,a1.value as C,a2.value as D
FROM eav_table_entity a
LEFT OUTER JOIN eav_table_attibute_value as a1
 ON a1.id=a.id
 AND a1.name="C"
LEFT OUTER JOIN eav_table_attibute_value as a2
 ON a2.id=a.id
 AND a2.name="D";
```

Figure 4.7 – Join approach (Get)

4.3.2 The multiple join approach

To overcome the limitation of 61 joins in MySQL, we tested several other possibilities. A possible solution is to execute joins of joins, composed of maximum 60 tables. For instance, in the case of a 120-attribute table we will build back through two joins of 60 joins. We will not say much about this approach as it is identical to the join approach in theory. This solution solved the maximum-join-per-query limit on select but does not allow the user to use this select to define a view based on the query. The performance issue is also remain important as the execution time is quite long. We provide in Section 6.3 some performance comparison.

4.3.3 The coalescing approach

In our quest for performance, we propose here a different way to perform the pivot operation. Instead of joining multiple tables, we work here with one unique select statement that will perform the whole operation. Figure 4.8 presents this method that we called the “Coalescing approach”. This Figure is decomposed in sub-steps that are presented here after.

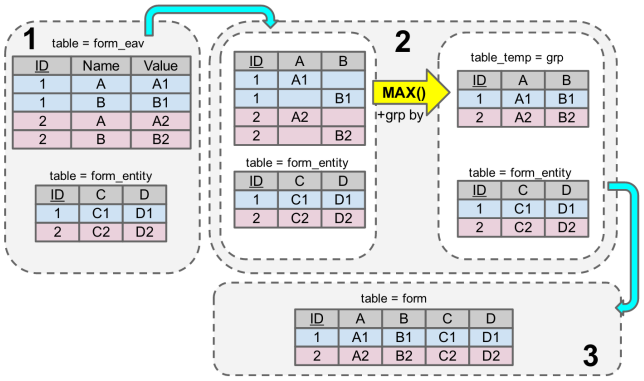


Figure 4.8 – The coalescing approach : general picture

The corresponding pseudo-code is given in 4.9.

```
SELECT id,A,B,C,D FROM
 (SELECT
  MAX(IF(a.name ='C',a.value,null)) AS 'C',
  MAX(IF(a.name ='D',a.value,null)) AS 'D',
  id FROM eav\_form\_attribute\_value a GROUP BY id) AS grp, eav_table_entity AS i
      WHERE i.id=grp.id;
```

Figure 4.9 – Coalescing approach (Get)

Although this operation is executed in one step, we can decompose it in three execution steps. First, the query performs a huge select, composed of if statements that will create a temporary table, as the one presented in Figure 4.10. This table contains all the attributes that have been pivoted as the attributes of the table (and one more attribute that identifies the entity). But only the id and one attribute are set for each row. The next step is therefore to keep the id of the entity and merge these rows to have all these values on one row per entity.

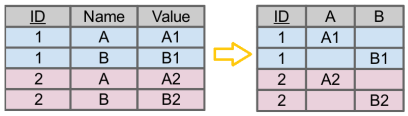


Figure 4.10 – The *select* in the coalescing approach

Then, in each of the if statement, the MAX function is used to "shrink" the table with a group by clause (for grouping on entities), as presented in step 2 in Figure 4.8 . In case there is no value for a given field, the if will introduce a NULL value. The result of this operation is presented in 4.11.

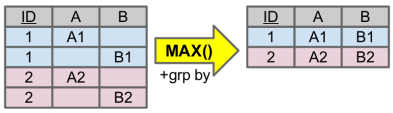


Figure 4.11 – The *max* in the coalescing approach

Once this is done, the original table has nearly been recreated. By joining this with the entity table, we will merge the entities (attributes kept in the original format) and the previous temporary table. Finally, we recreate the original table through a view, containing all the data from the EAV model.

This query can be used to define a view, which is not the case with the joins of joins technique.

4.4 Implementation conclusions

In this chapter we presented how it is possible to implement the main operations in order to keep the virtual database synchronized with the actual database. We provided some implementation alternatives that could help the reader in further implementations and presented several techniques for implementing the same function. Some performance comparisons of those different alternatives are presented in Chapter 6 (Section 6.3) based on real tables from our case study.

In the next chapter, we introduce the tool developed to help migrating tables to an EAV model and present its architecture and main functions.

Chapter 5

Tool support

The absence of out of the box solutions for Channels generation brought about idea of developing a tool to facilitate the schema evolution process by automatically generating these Channels. We developed a plugin for DB-Main (www.db-main.eu), an interactive database (re)engineering tool developed by the University of Namur and its spin-off company Rever. So far, DB-Main offers rich support for database schema transformations, but does not generate Channels. Our plugin extends DB-Main with the capability of evolving database schemas based on the aforementioned transformations. DB-Main generates the database definition of the newly evolved schema as well as all the code for the bidirectional Channel that allows legacy programs to run on the newly evolved database. We have experimented different alternatives to implement the required transformations, particularly Pivot and Unpivot, as these operators are not provided as built-in primitives by most database management systems, or at least, not as we had to use it.

5.1 DB-Main plug in API

DB-Main provides a rich support for database engineering and reverse-engineering and is a solid basis for the integration of a schema evolution tool. It furnishes a Java interface, JIDBM, encompassing functions to access the DB-Main repository in read and write mode. JIDBM provides a Java API (Application Programming Interface) for DB-MAIN users and makes it possible to write applications using a pure Java API. Unfortunately, DB-Main has its limitations. It does not allow access to built-in functionality of DB-Main and it is therefore not possible to use the existing DB-Main transformations tool or to modify its GUI (Graphical User Interface). However it is possible to develop an external plugin integrating its own

interface that is executed over the DB-Main application.

JIDBM is built on a layered architecture. Figure 5.1 presents this architecture.

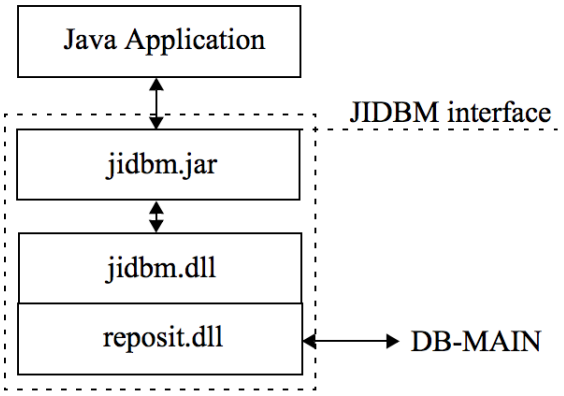


Figure 5.1 – DB-Main API Architecture (taken from [1])

The architecture is made of three main components :

- **reposit.dll:** a library that allows access to the main DB-Main repository.
- **jidbm.dll:** allows jidbm.jar (running within the JAVA Virtual Machine) to operate with reposit.dll.
- **jidbm.jar:** the java interface that can be used by the java application.

The use of the DB-Main API is transparent, users can instantiate a set of java classes allowing the creation and manipulation of DB-Main projects or DB-Main schemas directly. More information, including the DB-Main meta-schema and some code examples, are available in the official documentation. [1].

5.2 EAV Migration tool

Our DB-Main plugin can be used to pivot/unpivot tables from a DB-Main schema (SQL-DDL to DB-Main schema extraction is also available in DB-Main), one at a time or several at a time. The plugin also supports the migration of the existing data into the new EAV schema, generates the Channel implementation (triggers), generates the views, tests the data migration, etc.

The support provided by the DB-Main plugin is substantial for two reasons, *correctness* and *scalability*. The first reason is related to the safety-critical nature of health care information systems. There are strict requirements on the correctness of transformations and the ability for backwards compatibility of programs that use the "old" database structures. A tool that is capable of implementing schema transformations based on formally defined loss-less transformations as well as generating code for Channels that can be used to automatically adapt "legacy programs" gives welcome assurance in this domain.

Secondly, the size of the Channel code generated by our plugin is considerable. Writing this code by hand would be tedious and error prone. We found that in the best performing code (discussed below), each column in a transformed table gives rise to approximately 34 lines of code in the update (*put*) function of the Channel. A table of 1.000 columns will therefore give rise to 34KLOC of channel code for the *put* direction alone.

Before starting to generate the SQL code, create the triggers or migrate the data, it is necessary to create the EAV schema in the DB-Main repository. Figure 5.2 shows the manipulation steps needed to do so.

First, the database schema has to be imported into DB-Main (1). There are three ways to perform the manipulation:

- By importing the *.lun* file (DB-Main schema file format) into DB-Main
- By extracting the schema from the database
- By importing the SQL-DDL creation script of the database schema

Then, the user has to choose the tables to be migrated into the EAV model (2). If more than one table is chosen, the tables will be integrated into the same EAV schema. In this case, the tables must share common attributes in order to create the entity table.

In our EAV model, some of the attributes of the original tables are kept in the original representation and are stored in the entity table. The user can choose the attributes that will be stored in the entity table (3). When multiple tables are integrated into the same EAV schema, only the common attributes can be kept in the classical representation, sharing a common name and a compatible data type.

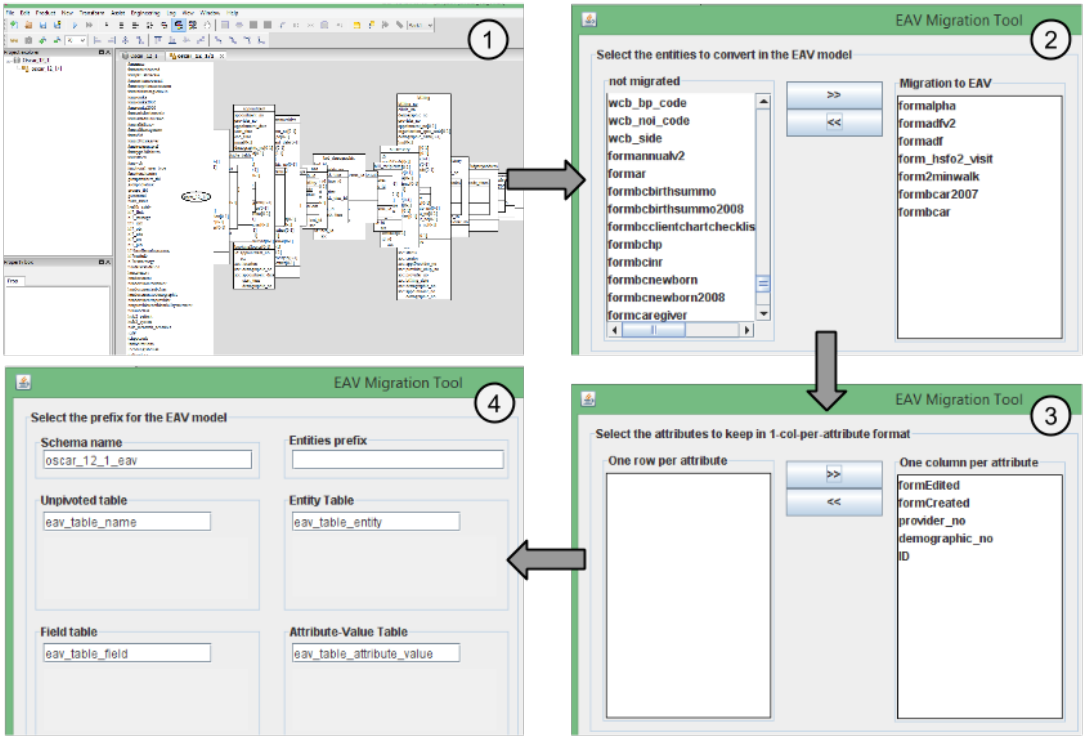


Figure 5.2 – DB-Main plugin: EAV model creation

The remaining attributes are therefore automatically stored into the EAV schema.

Finally, the user can choose the names for the different tables of the EAV schema (4). A name-check on the database is performed in order to ensure that there exists no table with the same name in the database schema.

Once all these steps are performed, the plugin creates the corresponding specific schema in the DB-Main repository (an example is given in Figure 5.3).

5.2.1 Tool functionalities

Once the DB-Main schema is created, several functionalities are available. Figure 5.4 presents the main functionalities of the plugin.

5.2.1.1 SQL-DDL code generation

The plugin generates the SQL-DDL creation script for the EAV schema. The creation script is saved, allowing the user to check it before executing it. The

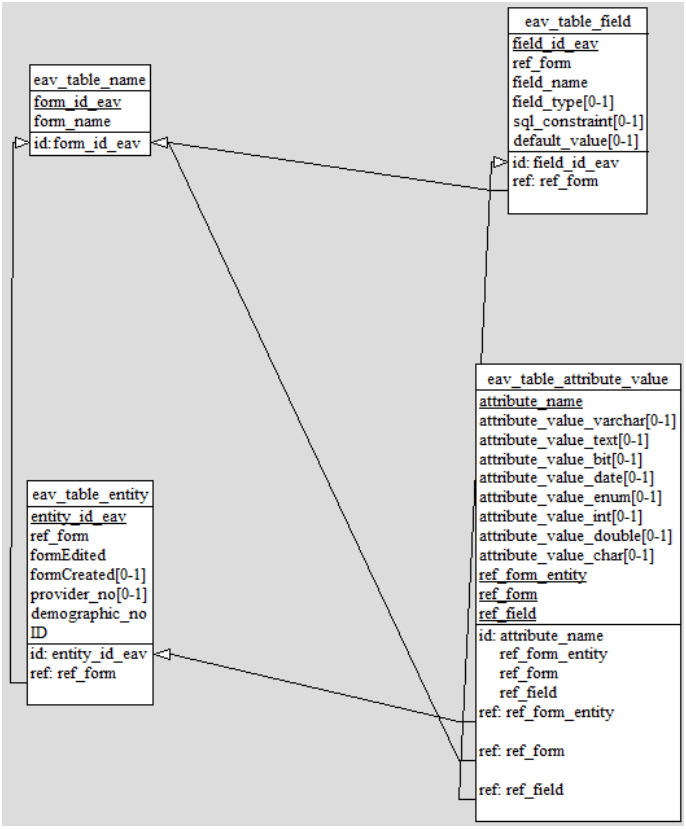


Figure 5.3 – DB-Main plugin: example of created EAV schema

SQL-DDL code is generated with a SQL generator for MySQL. Although DB-Main has a MySQL SQL-DDL generator, it shows to not preserve the original data types (from the original database schema) and to replace them by other data types (e.g. Boolean becomes CHAR(1)). Therefore, we reimplemented the MySQL generator for our plugin. The generated creation script is targeted to MySQL databases and is therefore not compatible with other DBMSs. By using other SQL-DDL generators, we can address other DBMSs.

5.2.1.2 SQL-DDL code insertion

The plugin allows the insertion of the previously generated SQL-DDL code into the MySQL database. In order to perform this insert, the user needs to specify the database connection parameters (see Figure 5.5).

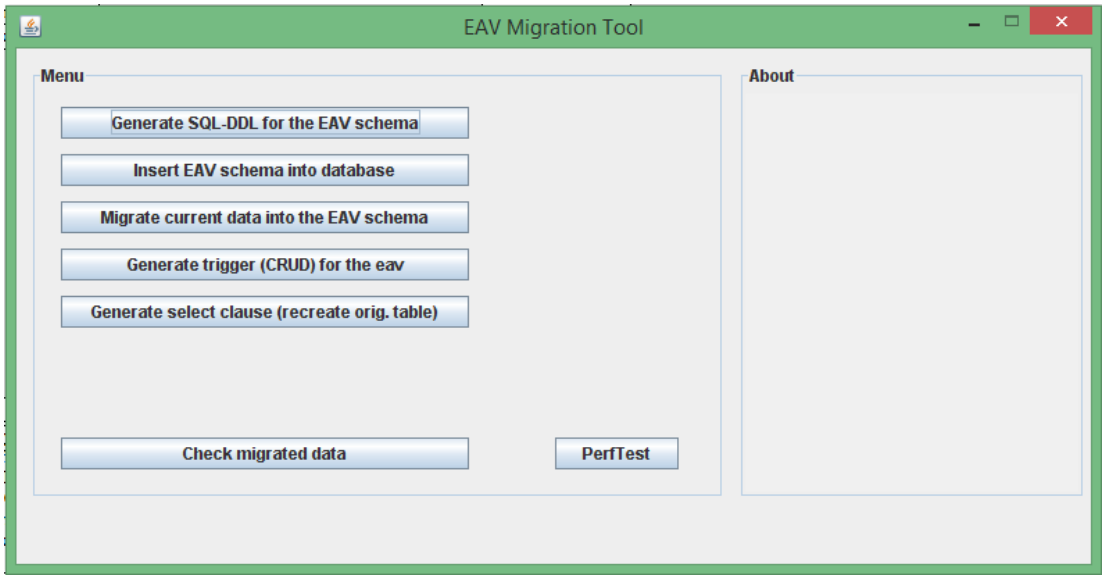


Figure 5.4 – DB-Main plugin: main functionnalities

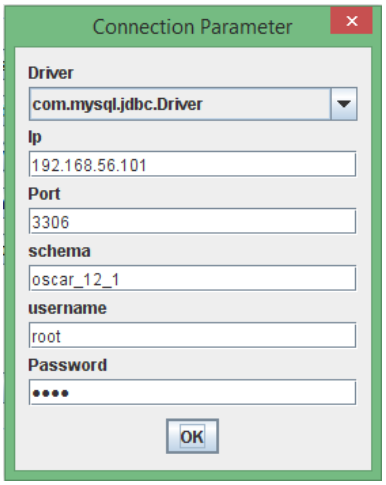


Figure 5.5 – DB-Main plugin: database connection parameters

5.2.1.3 Migration of the data

The plugin allows the data migration from the original tables into the created EAV schema. A dialog box gives the choice of three possible implementations to perform this data migration (see Figure5.6).

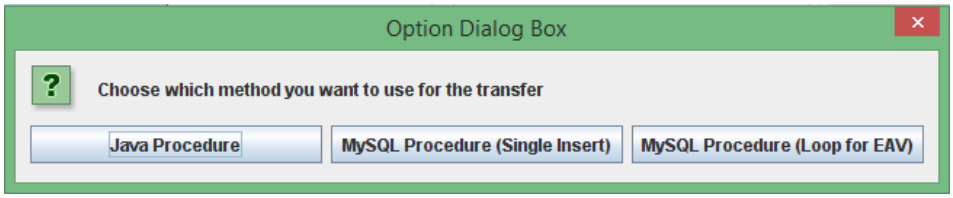


Figure 5.6 – DB-Main plugin: data migration implementations

The three different implementations to perform the migration are as follow:

1. **Java Procedure**: Data is migrated using a Java procedure that extracts the data from the database and reinserts it into the EAV schema.
2. **MySQL Procedure (Single Insert)**: A procedure migrates the data through a unique insert statement for unpivoting the table. This method refers to the coalescing approach presented in Chapter 4.
3. **MySQL Procedure (Loop)**: A procedure migrates the data by performing a loop on the records from the original table. This method refers to the procedural approach presented in Chapter 4

Each implementation has its advantages and its drawbacks which has been discussed in Chapter 4.1. In summary, the Java procedure is the slowest but has the benefit of being compatible with all the DBMS as long as the proper JDBC driver is loaded. The single insert MySQL procedure is the fastest but in case of very large data set, it can face some issues if the database is not properly parametrised for handling large amounts of files. Finally the MySQL loop is a good compromise, it is not as fast as the single insert but it does not have the problems of the single insert.

Regardless of the migration implementation, a script is created (in case of database side migration), inserted into the database and then executed. The user can select tables that have to be migrated (Figure 5.7). As we wanted the tool to be generic, we allowed the user to create an EAV schema containing multiple tables, with the option of migrating each table when appropriate.

5.2.1.4 Trigger generation

The tool can also generate triggers to perform the *put* function. The triggers are created but not automatically inserted, the code is saved in order to allow code control before insertion. The trigger generator is developed for MySQL but is also

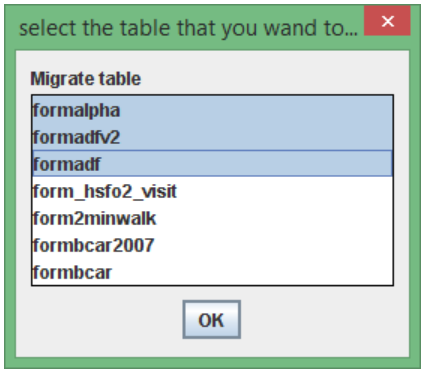


Figure 5.7 – DB-Main plugin: tables selection for data migration

compatible with Oracle DBMS. Due to MySQL limitations, "instead of" triggers cannot be used on the views that recreate the source table. However, they can still be used on physical tables (to keep an EAV schema up-to-date in order to plan future evolution). Other DBMSs support "instead of" triggers on views (Oracle for instance).

For each table, three triggers are generated: one for the insert operation, one for the update operation and one for the delete operation. Their implementation has been presented in Section 4.2. The size of these triggers and their complexity (especially in the case of the update trigger) make them difficult to develop manually. Generating them automatically is therefore a good way to avoid error and save time.

5.2.1.5 View generation

Views are used to build back the original representation of a table that has been migrated into the EAV schema. The tool allows generation of these views with SQL-DDL creation scripts. Three different approaches have been developed (see Section 4.3) and are listed below. Performance characteristics of each implementation are discussed in Section 6.3.

1. **Left outer join (with max join per query limit)**
2. **Left outer join (without max join per query limit)**
3. **Coalescing approach**

The max join per query limit has been addressed in Section 4.3. The limitation is directly dependent on the DBMS. On MySQL the limit is set to 61 joins per query,

but for MS-SQL Server 2005, the limitation reaches 256 joins and does not exist anymore in MS-SQL Server 2008 (the DBMS is only limited by available resources).

5.2.1.6 Migration validation

After performing the data migration, it is important to check the validity of the migrated data. The tool allows to check the validity of the migrated data by comparing the original tables data with the recreated tables (using views) on top of the EAV model. In case the application detects an error, it warns the user with a message. (see Figure 5.8).

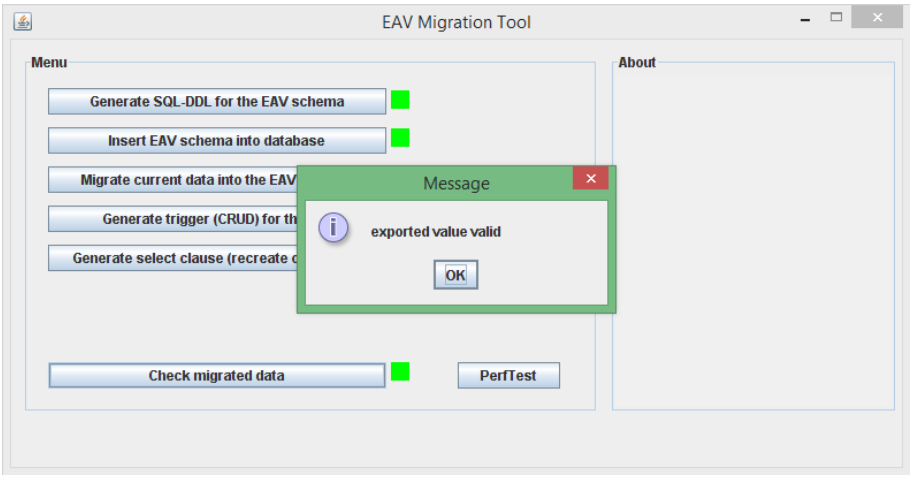


Figure 5.8 – DB-Main plugin: migrated data validity check

5.3 Benefits of the tools

As discussed, this tool provides many functions to support the concrete Channels implementation as well as data migration. Many of these functionalities avoid manual code development and thus prevent errors. As most of the generated code is dependent on the number of attributes, triggers and view creation code can easily reach thousands of lines of code for large tables.

According to user's needs, this tool can be used in many scenarios. It can perform a complete migration of the database allowing new software to use the EAV schema while legacy software continues working with views (allowing backward compatibility). It can also be used to create an EAV schema and migrate the data

while developers perform a manual code refactoring. In this case, the EAV schema can be created in advance and kept up-to-date thanks to the put function. This solution allows to shorten the downtime during the effective migration.

In the next chapter, we present a concrete real-world case study based on the OSCAR application on which we applied the previously presented techniques and tools.

Chapter 6

Case study : OSCAR

In this chapter we present the application case study used in our work. The OSCAR EMR system is presented first, followed by the methodology used through this research, some performance measurements that have been taken and finally, we explore the possibility of using code-refactoring in this concrete application.

6.1 The OSCAR system

OSCAR (Open Source Clinical Application Resource : http://OSCAR-emr.com/) is an EMR (Electronic Medical Record) used in primary health care in hundreds of clinics in Canada. It is a software developed to support doctors and clinical practices. OSCAR provides all the necessary functions to run a medical practice/-clinic [30], in which we can mention: patient registration and scheduling, billing, prescription and so on.

OSCAR development started in 2001 in McMaster University (Hamilton, ON, Canada) to become, nowadays, one of the most used EMRs in Canada. It is supported by a strong community of doctors and companies that provide technical support for the clinics [30].

Also, OSCAR is FLOSS (Free libre open source software) under the General Public Licence (GPLv2), which ensures that OSCAR is available with no licensing fees [30]. Developed by doctors for doctors, the system is under constant evolution due to its large amount of collaborators through the world. One of its strengths is the variety of auxiliary software that provides multiple supports. For example, MyOSCAR that provides personal health record on-line consultation for the patient, or MyDrugRef that promotes exchange of information on drugs between practitioners for better

medical prescriptions and so on.

With the increasing amount of data flows created by these EMR, collaboration between inter-domain universities is born. New industrial techniques of data mining are now going to be applied on medical data in order to optimize patient care. For this purpose, the integration of several EMR data in one central repository is necessary and to accomplish this task, many research laboratories are now working on the integration of EMRs.

In the Simbioses Lab, doctors and computer scientists are working together to achieve the goal of creating a global repository able to provide data mining capacities. Their project, called SCOOP, is currently under development and has already been presented to several clinics that are highly interested by the new tool's capacities.

In order to integrate the EMR data, some systems have to evolve in such a way that their data structure is more convenient for the SCOOP repository. The aim of this case study is therefore to show how the application of our database schema migration techniques has helped in this process.

6.1.1 OSCAR architecture

The OSCAR EMR is developed under a classical 3-tier architecture. The data management is based on a MySQL Server instance running the database both on InnoDB and MyISAM storage engines. The application itself is running under Apache Tomcat and uses a large set of various java technologies. Finally, the client used by practitioners is a web interface providing all the needed operations for an every day use.

For accessing the data, the OSCAR system uses many different technologies. As part of an open-source community, the project tries to set guidelines but many parts of the application integrate new technologies or still use legacy ones. The oldest parts of OSCAR use a Java Database Connectivity (JDBC) and query the database directly. In the newer portions, this technique has been replaced by an Hibernate framework (an Object Relational Mapping framework) using XML files to define the mapping. Nowadays, in the most recent development and also in the OSCAR guidelines, Hibernate is still used, but the mapping has to be set with JPA

annotation instead of the XML files.

We can easily see in this fragmentation the consequences of a slow evolution of the code without a major code re-factor. Moreover,the geographical dispersal of programmers and the different methodologies and approaches they use are surely responsible for leading the project to the existing application heterogeneity.

6.1.2 OSCAR database schema

The OSCAR system has evolved over more than a decade and its current database includes more than 450 tables. The OSCAR database consists of well-populated "core" table structures that store information about patient demographics, allergies, medications, active problems etc., as well as more specialized, "satellite" table structures that store information for specific types of encounters and patient situations in primary care. These more specialized table structures are often associated with elaborate electronic *forms* that are filled out by the clinician on certain types of patient encounters. Due to the broad spectrum of different conditions that patients may have, these tables may have thousands of columns but any given data record (the actual data in each row) may only be populated sparsely (i.e., many `null` values).

Although the state of the application is not really problematic for its current use, it hinders the system's integration with some data mining applications such as the SCOOP project. Also, the actual schema is quite complicated and difficult to understand. Providing a database schema migration would definitely help giving a clearer overview of the database to programmers, impacting side effects in the code quality and would also allow the OSCAR EMR data to be integrated into other systems. Being a commonly used EMR in Canada, OSCAR is a tremendous source of available information for data mining purposes.

6.1.3 Database and program co-evolution

In order to allow the integration of the OSCAR data into other systems, we had to evolve the database schema of the application. Deriving from the evolution of the database schema, the link between the application and the database had to be reconstructed and this is where our designed solution takes place. By using

our solution, we aim to provide a translation layer that simulates the behaviour of the legacy database schema, while using a new physical structure implementation.

Figure 6.1 illustrates the classical database and program co-evolution problem. The reader can see that legacy applications make use of the old database while newly developed applications can take the full benefits of the new database. By providing our intermediate layer, the Channels implementation, we therefore allow legacy applications to use the new database.

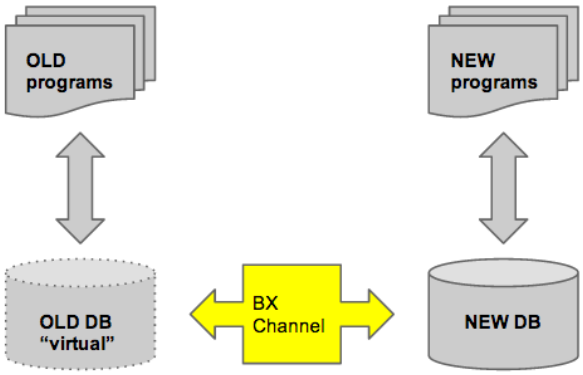


Figure 6.1 – BX in DB/program co-evolution

The section 6.2 presents how we proceed in this research in order to apply this migration to the existing database.

6.2 Methodology

This section presents the methodology adopted for this case study. We present here the different steps we followed in order to meet the previously fixed objectives. Planning and executing the migration of the OSCAR forms related tables is only a part of the work and several steps had to be accomplished first.

6.2.1 Documentation process

The first phase of the internship was documenting the software and the database architecture. This step is mandatory since evolving software implies having a good comprehension of it. We defined two objectives for this documentation step. The first was to identify all the forms contained in the application and get information about their data structure. The second was to understand how the software queries

the database in order to identify possible alternatives for the database schema evolution.

We first tried to retrieve as much existing documentation as possible. Unfortunately, with the exception of a deployment guide and some developers guidelines, the documentation on OSCAR is nearly non-existent. We therefore went through a reverse-engineering phase.

As it was not the main subject of our research, we wanted to keep this step as simple and brief as possible and focus only on the subsets of interest. In that way we avoided an heavy software assisted reverse-engineering process such as software slicing and so on. On the database side, apart from using works of the previous interns [31], we only used manual data reviewing. On the software side we used manual code reviewing supported by the tools provided by Eclipse (mainly the search engine and the call hierarchy retrieval). These tools and techniques have also been used for the three following phases of the reverse engineering.

6.2.1.1 Form structure identification

We first searched for documents that identify the forms. As we did not find this information, we tried to find the forms related source code and the related database tables. We identified a large amount of tables with names matching the pattern "form[a-z|A-Z|0-9]*" we first made the assumption that many of them were actual forms. But as we needed a formal identification we started to look into OSCAR application code for an artifact that could have helped us. After some research, it emerged that there was a table containing all the forms names and the actual table name corresponding to each one. We counted 63 different forms.

6.2.1.2 Analysing the forms

A manual reviewing of the forms related tables showed that a common structure was used between all the forms. Every form contains five basic columns representing the unique identifier of each entity. One of these columns, the "demographic_no" field is also an implicit foreign key. The demographic_no is a numeric value of 8 digits that formally identifies a patient.

There is no existing constraint defined on the schema up to now, except for primary keys. The constraints are not declared in the database. Indeed, depending on the

form table, the “demographic_no” can be of several types (int, bigint, varchar) and have different lengths (10, 11, 20). This makes impossible to create a foreign keys constraints between tables. This information is useful to choose which attributes can be pivoted and which ones have to remain in the standard format.

6.2.1.3 Understanding the data access layer

The final step of this reverse-engineering phase was to understand how the software communicated with the database. This information was vital to plan the evolution. According to the available documentation there are three different means by which OSCAR accesses the database:

- DBHandler: an old database connector that directly queries the database (using JDBC).
- Hibernate: the object relational mapping framework using XML mapping files.
- Hibernate/JPA: Hibernate extended with the use of JPA annotations.

Now, the development guidelines ask one to use the JPA mappings but there are still many parts of OSCAR that use other ways of accessing the database. We wanted therefore to know which of these technologies are used by the forms in order to plan possible software evolution. Since the OSCAR package structure was properly made, it was not too difficult to find the corresponding classes. It appears that the forms use the old database connector in a flexible manner.

Considering that refactoring the code for all the forms to make use of the ORM was highly time consuming, even more since there is no possibility to express complex transformations as the one used in this work with ORMs mappings, we decided not to use ORMs for this migration. The remaining possibilities to use the new database schema were :

- To build backward wrappers
- To automatically evolve the software code.

After considering the two possible solutions, we decided to automatically build backward wrappers. In fact, automatic evolution of the code to make use of ORMs would have been the best solution but, due to the internship time limitation, this solution has not been investigated.

6.2.2 Structure and data migration

Data gathered during the reverse-engineering phases was used to build our new database schema. After designing a custom and type-safe EAV model, we had to migrate all the data into this structure. The migration was performed using the implementation of Channels presented in Chapter 4.

6.2.3 The Channels implementation

We started with the *create* function which is used to migrate the data from the original tables into the EAV model. Then, we continued with the *get* function which is used to create the view. Finally, we developed the *put* function, that is used to maintain the schema synchronised with the views. After manual proof-of-concepts, we implemented a tool to automatically generate these Channels.

6.2.4 The migration tool

The development of a tool supporting Channels creation was a key point in our work. The migration development followed a prototyping life-cycle. The first prototype was developed during the creation of the first Channel. This one had no user interface and restricted functionality and expressiveness. We then migrated this implementation as a case tool plugin and extended its features.

6.2.5 Performance tests

In the last part of this internship, we performed some performance tests on the *create* and *get* functions. The results of these tests are available in Section 6.3 of this chapter.

6.2.6 OSCAR code refactoring

To conclude, we implemented a real code refactoring of OSCAR that uses the new data structures directly, without wrapper usage. This refactoring was made with the idea of generic use in mind and we developed some classes that can be used to facilitate the forms migration. Making use of these classes facilitate the code refactoring. All the forms classes access the database by means of one main data access layer (DAL). We therefore provided a refactoring of this DAL. This refactoring is discussed in Chapter 7.

6.3 Performance analysis

In order to evaluate the viability of our solution in a real world application, we decided to conduct some performance tests. We provide below some performance measurements of the different implementations presented in Chapter 4.

6.3.1 Foreword : the data generator

Medical information systems manage a lot of information that must remain private. This makes it difficult to obtain real data for testing purposes. In some cases, after a motivated appliance and a long justification process, it is possible to obtain an anonymised database in order to perform some tests. But this appliance can take several months. We unfortunately did not receive it in due time and as such, we decided to develop a data generator.

The data generator simply gathers the table structures and generates random values to instantiate the database schema. Considering the potential sparseness of the form tables, we implemented a feature allowing the definition of a the sparseness percentage in the generated data. We then used generated data in order to verify the applicability of our solution and to provide some performance comparisons.

6.3.2 The create function

First, we present a comparison of the performance for the data migration from the original model to the EAV format. This step corresponds to the implementation of the *create* function. It is composed of a VPartition operation followed by the Unpivot operation. For these performance tests we decided to benchmark the data migration time on three different tables. We chose three tables from OSCAR containing 17, 117 and 425 columns. The following chart gives an overview of the time needed to migrate 1000 records from the original table to the EAV table. Figure 6.2 shows the different performance characteristics of the declarative implementation and the procedural methods.

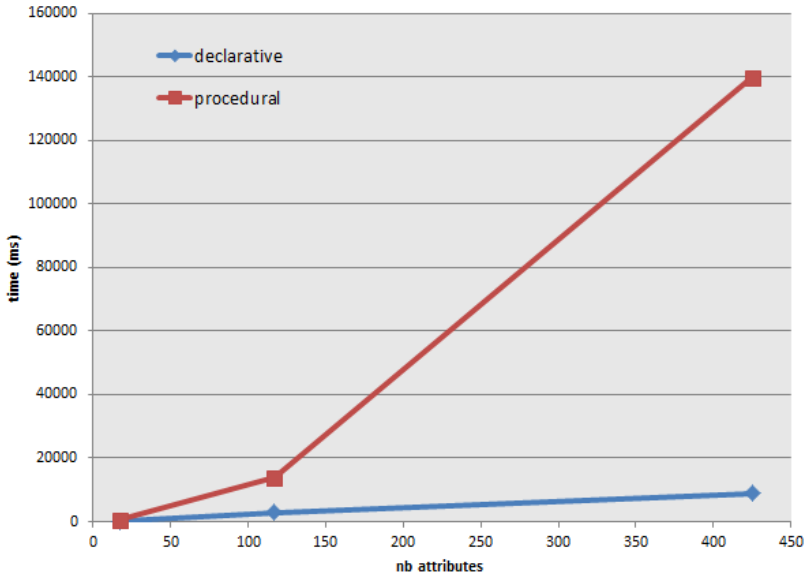


Figure 6.2 – *Create* performance.

The performance gap between the two unpivoting methods can be understood by taking a look on the SQL query. The procedural statement executes a query for each value of each line that has to be unpivoted to insert all the field value one by one. The DBMS query optimizer is not able to optimize this loop. On the other hand, the declarative approach executes only one insert query. This query is composed of one sub-query per attribute, but is not directly dependent on the number of rows contained in the original table, even if it will impact the data set size. The DBMS query optimizer can optimize and execute this single query more efficiently.

6.3.3 The get function

We also took measurements on the view reconstruction query (*get*). First, a Pivot operation is executed and then a VMerge is applied on the result of the Pivot operation with the table that contains the attributes kept in the "classical" relational form. We present in the Figure 6.3 the time to pivot/merge the table for the join approach and the coalescing approach. For these measurements we took the same tables as above and measured the time needed to perform a select query on the EAV model containing 1.000 entities.

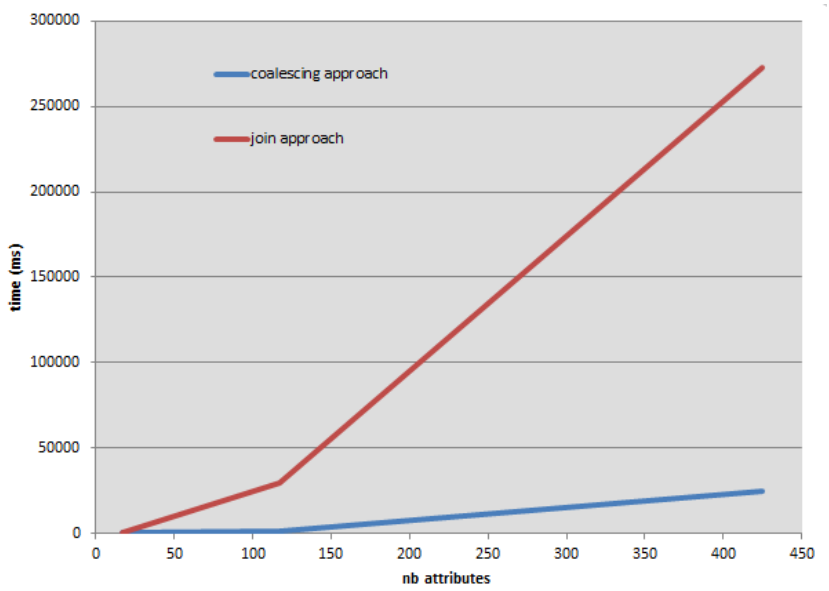


Figure 6.3 – *Get* performance.

Comparing the performances of the two pivoting methods, we see that the first method uses a lot of joins (costly database operation), by creating one temporary sub-table per pivoted attribute. The second approach (coalescing approach) performs a unique select query that retrieves a huge result-set, then manipulates it to pivot the data. This method performs no join nor any other costly operator. It only uses a single select with some conditions and is therefore faster.

6.3.4 Table size impact

We expected that the number of migrated records also had an impact on the query response time. We therefore migrated 50, 100, 500 and 1.000 records from a 425-attributes table of 425 attributes in order to see the time needed to compute the view. Both coalesce and multiple join implementations have been tested.

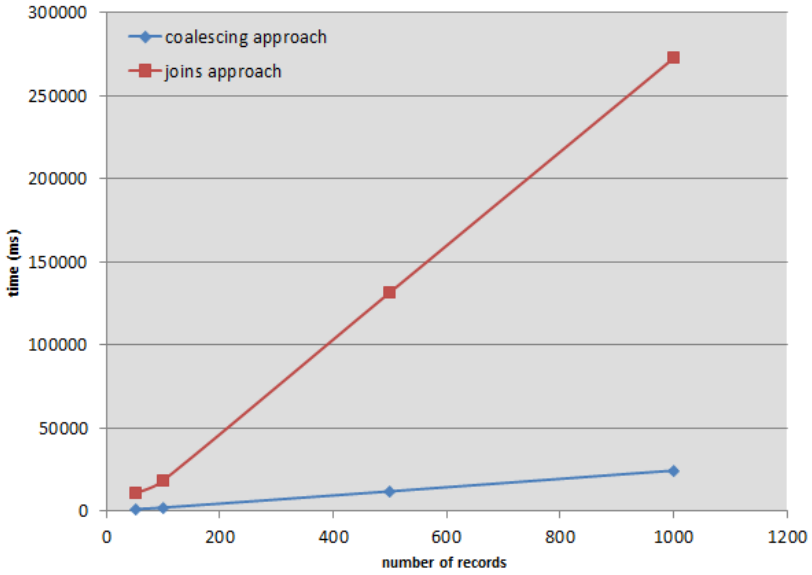


Figure 6.4 – Number of rector impact.

Figure 6.4 shows that the query response time rises linearly with the number of attributes of the table. As it takes 2s to pivot 100 records, it takes about 24s to pivot 1.000 records.

6.3.5 Data sparseness impact

The EAV model is frequently used to avoid data sparseness in rational database. Therefore, performing only benchmark on non-sparse tables may not be representative of the real context and so, we performed some benchmarks of the *get* function with various degrees of data sparseness in order to evaluate the impact of the data sparseness.

The form "formRourke2006" which contains 627 attributes was chosen for this evaluation. Three test cases were defined in which tables always contained 1.000 rows and a varying occupancy rate of 1%, 10% and 100% (no data sparseness). The *get* function was defined by using the coalescing approach as it is the fastest implementation. Figure 6.5 presents the performance comparison.

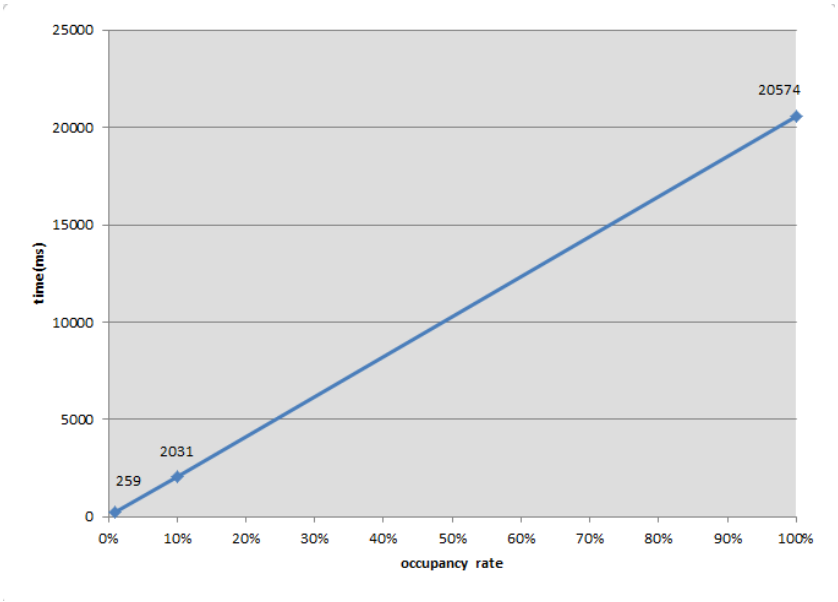


Figure 6.5 – Sparseness impact on coalesce implementation.

Figure 6.5 shows that the impact of data sparseness is linear on the query response time. Indeed, as the *null* values are not present in the EAV table, the sparseness impacts the number of row to merge (see Section 4.3) and consequently the response time of the query.

6.3.6 Query optimization and the Prune Level

Although the performance of the EAV model and the view reconstruction is quite effective for practitioners, it is interesting to take a closer look at how it works inside the DBMS.

Although most of the time forms contain very sparse records, it could be interesting to see how MySQL manages the view reconstruction and selection on it. The point is that, when practitioners use their EMR software, the program queries the view, mostly to retrieve only one record corresponding to a specific patient for a specific form. Therefore, it is interesting to analyze the execution plan of a query executed on the view in this case.

The question can be summarised as: "Does MySQL compute the complete view and select only the records corresponding to the given ID ?" or "Does MySQL select the entity and thereafter join it with the subset of rows from the EAV table

corresponding to the same entity?"

In a more formal way, the question is : "Does MySQL work as 1 or in 2 when selecting on the view based on an ID" ?

1. $(A_{ID} \bowtie B_{ID})\sigma_{ID=\mathbf{X}}$
2. $((\sigma_{ID=\mathbf{X}} A)_{ID} \bowtie A_{ID})$

After analyzing the MySQL queries plans in the query optimizer (using the function "Explain Extended"), MySQL was found to compute the view in its totality and then, the selection is executed on the computed view. According to the official MySQL documentation, modifying the "prune level" could allow the database engine to take the number of rows used in the join into account. It can result in a performance problem for the overall system as the compilation of other queries could be a lot slower than before.

By default, the MySQL engine is not set to take the number of lines into account. This parameter can be changed for purpose of database tuning, which is the case here. After an analysis of the query plan when executing a select on a form view with the prune optimizer variable set to 0 (taking number of rows into account), the database engine did not change the query plan and the execution time was exactly the same.

By analysing the view definition, one can notice that it includes a "group by" clause and aggregate functions (MAX(...)). The use of this clause unfortunately makes the query optimizer fail when attempting to compute only the needed join, as defined in the first formal definition before. So, for each select, the view has to calculate the total join for every entity_id and finally, only the row related to this entity_id will be selected. We therefore investigated another solution to compute only the needed join, as defined in the second formal definition before.

After some research, MySQL showed no to support parametrized views but in this case, a simple implementation alternative solved this limitation. By using this "parametrized view", we drastically decreased the response time when selection on the view. Here after, we present how parametrized view can be defined.

First, a function to store the entity_id as a variable has to be defined. Figure 6.6 presents this function declaration.

```
create function p1() returns INTEGER DETERMINISTIC NO SQL return @p1;
```

Figure 6.6 – Parameter function definition

Then, the parametrized view is defined using the variable from the previously defined function.

```
CREATE VIEW formrourke2006view AS
SELECT i.ID, i.demographic_no, i.formCreated, i.formEdited , i.provider_no,
MAX(IF(attribute_name ='attr_1',attribute_value ,null)) as 'attr_1',
MAX(IF(attribute_name ='attr_2',attribute_value ,null)) as 'attr_2',
...
MAX(IF(attribute_name ='attr_n-1',attribute_value ,null)) as 'attr_n-1',
MAX(IF(attribute_name ='attr_n',attribute_value ,null)) as 'attr_n',
ref_form_entity FROM eav_form_entity i, eav_form_attribute_value grp
WHERE i.entity_id_eav = grp.ref_form_entity
AND i.ID = p1()
AND i.ref_form = (select form_id_eav from eav_form_name where form_name='
    formrourke2006')
GROUP BY ref_form_entity;
```

Figure 6.7 – Parametrized view definition

Finally, the view parameter has to be defined and then the query can be executed. Figure 6.8 presents the query allowing to define the parameter and execute the select on the view.

```
select * from (select @p1:=10 p) param , oscar_12_1.formrourke2006view;
```

Figure 6.8 – Selection on the parametrized view

The respons time of this implementation alternative is nearly as low as classical select. A select on a table having about 1.000 columns is still computed in less than 100ms. We present in Figure 6.9 some time response comparisons when selecting one entity on the view.

6.3.7 Conclusion on performance analysis

Does this wrapper-based implementation fulfil practical user needs in terms of response time? The answer is definitely yes. Although the original coalescing

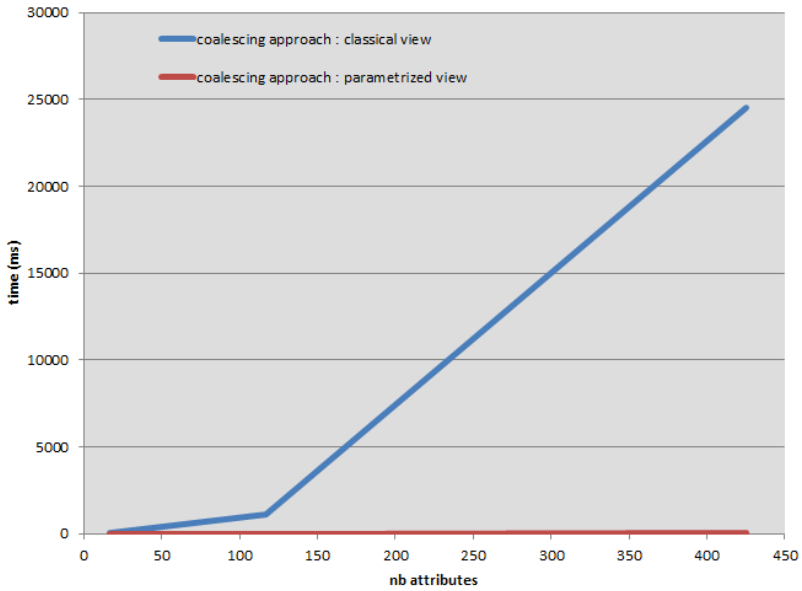


Figure 6.9 – Coalescing approach : Classical view VS Parametrized view

approach does not show sufficient performances for a practical use, using the parametrized view solves the performance issues. The time to select is now close to the select time on the original materialized table and therefore, we do not see any practical issue that would prevent the use of this technique in a real work application.

Chapter 7

OSCAR program code refactoring

The previous chapters presented the design and the implementation of the OSCAR forms module migration through auto-generated SQL wrappers. Those wrappers aim at reconstructing the view of the original relational schema from a specific EAV model. This implementation, although theoretically correct according to the SQL standard (SQL:2003), is not usable as such in our real context. In fact, depending of which DBMS is used, limitations may appear. Therefore, some parts of the work are not directly applicable as such.

In the specific case of OSCAR, the database engine is MySQL. MySQL is now an Oracle property but does not implement some important and necessary functions and operators. More than that, few DBMS implement the whole SQL standard. Today, SQL:2003 is still considered the current standard although more recent revisions exist.

7.1 Forms stability through releases

Before presenting the MySQL DBMS limitations that affect the OSCAR code refactoring, we would like to say a few more words about the OSCAR forms and their evolution through the software releases. The forms are ageing constructs from the OSCAR EMR but they are still used on an day-to-day basis by practitioners. This daily usage makes it impossible to remove it from the OSCAR release and as such, the only option is to migrate those forms to another database structure.

Today, there are about 60 forms related tables in the OSCAR schema considering release v11.Out of those 60 forms, 10 forms are specific to the British Columbia

release. Therefore, only forms shared by all the OSCAR versions (British Columbia and Ontario) are considered in this section. As depicted in Figure 7.1, the number of forms in the OSCAR releases seems relatively stable since schema version 120. Between version 120 (date: 2006-10-17) and version 670 (date: 2013-06-27), there have been approximately 7 years. During this time only 5 tables were added to the database schema and no forms were added since 2010.

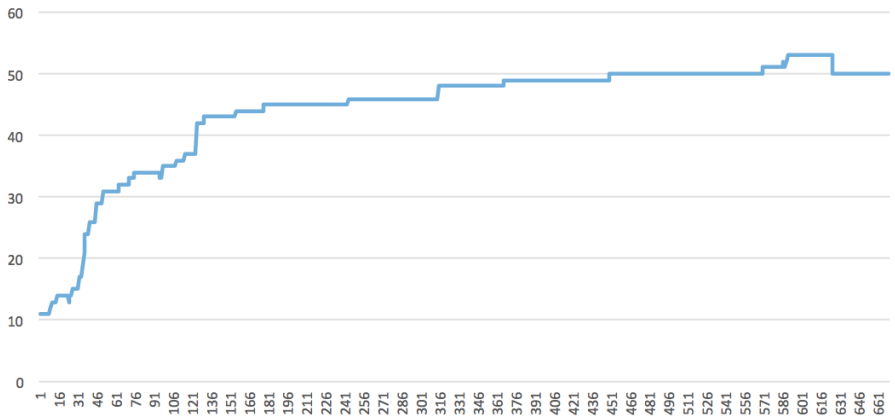


Figure 7.1 – Number of forms through OSCAR release

This stable evolution is explained by the fact that forms are being replaced by Eforms (Electronic forms using a combination of image and Javascript). However, considering that those forms are still used by practitioners on a day-to-day basis, their migration is mandatory.

It can be said, without taking too many risk, that the migration of those forms is facilitated thanks to their stability through the releases and the consideration that forms are being replaced by EForms. The next section presents the MySQL limitations that imply the need for the OSCAR program code refactoring.

7.2 Reasons for the code refactoring

Earlier, we made a parenthetical comment on the absence of Pivot and Unpivot operators in the MySQL operations set. Here, we present some other limitations that lead us to the actual OSCAR program code refactoring.

7.2.1 The view-update problem

The view-update problem is well-known and appears when users need to update the data sources through views. This concept has already been presented in Chapter 2. The view-update problem is exactly the problem to solve in this case.

In the next sub-sections, we present different possibilities before rejecting them because of technical limitations. Finally, another implementation alternative through program code refactoring is presented.

7.2.2 MySQL triggers limitations

The trigger mechanism is well-known in the database domain. A trigger is composed of procedural code that is automatically executed in response to certain events on a table or a view of a database. However, the specifications and the possibilities that triggers offer depend on the DBMS, in this case, MySQL. For instance, one limitation of this DBMS is that it is not possible to create a trigger on a view. In fact, the documentation is clear on this point. *("You cannot associate a trigger with a TEMPORARY table or a view.").*

To face this limitation, another possible solution is to use triggers but, instead of using a classical view, to keep the original table as a "materialized view" synchronized with the EAV model. The triggers, then applied to a "real" table (the materialized view), were allowed and could be used to propagate the changes from the table to the EAV model.

The problem with this solution is that it keeps all the original forms tables in the schema, introducing even more table structures than before, but also, MySQL does not allow creation of "instead of" triggers and so, "before insert" or "after insert" triggers must be used.

Due to the number of limitations in MySQL, the decision has been taken not to use triggers in the final implementation. However, on other DBMS, this solution works efficiently and thus, given the current discussion on a possible DBMS change in the OSCAR community, this concrete implementation is still highly interesting.

7.2.3 MySQL updatable views limitations

Another explored option to solve this view-update problem is to use updatable views. Updatable views allow, when the data is updated in the views, a propagation of these changes to the original data sources. Once again, depending on the DBMS, the view-update mechanisms may be different.

In the specific case of OSCAR, it is not possible to use updatable views for multiple reasons. In fact, MySQL allows some views to be updatable, depending on some constraints. The MySQL official documentation says the following :

> *Some views are updatable. That is, you can use them in statements such as UPDATE, DELETE, or INSERT to update the contents of the underlying table. For a view to be updatable, there must be a one-to-one relationship between the rows in the view and the rows in the underlying table. There are also certain other constructs that make a view not updatable. To be more specific, a view is not updatable if it contains any of the following:*
>
> - *Aggregate functions (SUM(), MIN(), MAX(), COUNT(), and so forth)*
> - *DISTINCT*
> - *GROUP BY*
> - *HAVING*
> - *UNION or UNION ALL*
> - *etc...*

It is clear that, considering the queries for the view definitions presented in Section 4.9, it is not possible to use updatable views in our specific case. Considering the limitations in MySQL, we have to find another way to implement a solution for the view-update problem in this specific case. The following section presents a hybrid solution using views and program code refactoring.

7.3 A hybrid solution : views and code refactoring

7.3.1 The forms module

In order to proceed to the actual OSCAR code refactoring, the first step is to understand how the form module works. Fortunately, despite the lack of documentation, the forms related Java classes are easy to understand. All the forms related classes are grouped into a package.

7.3.2 Data access layer and Java classes

The current structure of the forms module can be decomposed into three different parts, each of them assuring different functionalities.

- **The forms Java classes:** Each form is related to a specific Java class. This class implements some operations as retrieving a specific form instance, saving a form instance, etc.
- **The database communication class:** the DB-Handler class allows the communication with the database. This class allows to directly execute SQL statements using classical Java Statements.
- **An intermediate layer:** This layer is composed of a Java Class called *frmRecordHelp*. This class groups the common operations for all the forms and is an intermediary layer between the form Java classes and the DB-Handler.

Figure 7.2 presents how those components interact in the system. When the system has to save or retrieve a form instance, the associated form Java class is instantiated to create a form object. Operations on this form can then be directly executed by invoking methods from the form class on the previously created object. Each form implements the same basic functions but provides a specific implementation for several methods. When creating a new form, a new Properties object ($java.util.Properties$) is created, storing the minimum data needed for the form. Then, the Properties object is filled with the form values entered by the practitioner on the system. After that, the Properties object is given as parameter to a function (saveFormRecord() from the *frmRecordHelp* class) which, by invoking the DB-Handler functions, takes care of saving the form data into the database. When retrieving a form, function getFormRecord() from the *frmRecordHelp* class

fills the Properties object with corresponding data in the database.

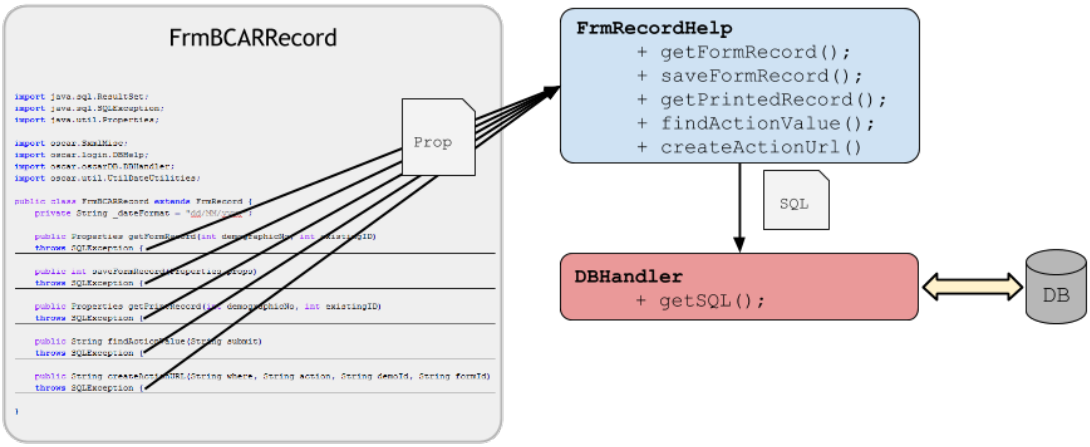


Figure 7.2 – Original Data Access Layer

A particularity of the original data access layer or DAL (the *frmRecordHelp* class) is that the system only executes select queries and does not execute any direct update or insert queries on the database. Instead, Java allows to update a ResultSet object (*java.sql.ResultSet*). Therefore, in order to insert or update data in the database, the system executes a select query with a never-satisfiable condition allowing it to obtain an empty ResultSet containing the table structure. The ResultSet is then updated with the form data and by invoking a function on this ResultSet object, the data is inserted or updated in the database.

Unfortunately, this clever implementation cannot be used anymore as tables are replaced by non-updatable views. Consequently, we had to provide the program code allowing to directly query our new database structure.

Considering that the views are still usable for retrieving forms data (all the manipulation in select mode), we provide a partial program code migration in which we replace the original *frmRecordHelp* class by a similar class allowing to query the EAV model. This class is called *frmRecordHelpEAV*.

For this refactoring, we wanted to have the lowest possible impact on the original program code in order to facilitate the code migration and even to leave the possibility of an automated migration of the forms module. By providing a compatible data access layer implementation (*frmRecordHelpEAV*) and providing function im-

plementation specific to the EAV model, the only needed operation on the original form classes is to redirect the function calls from the original DAL to this new DAL. Figure 7.3 shows the refactored architecture of the form module. As the views are used for all the retrieve operations, the original DAL is still used by the forms. However, when executing insert or update operations, the function calls have been redirected to the newly created DAL.

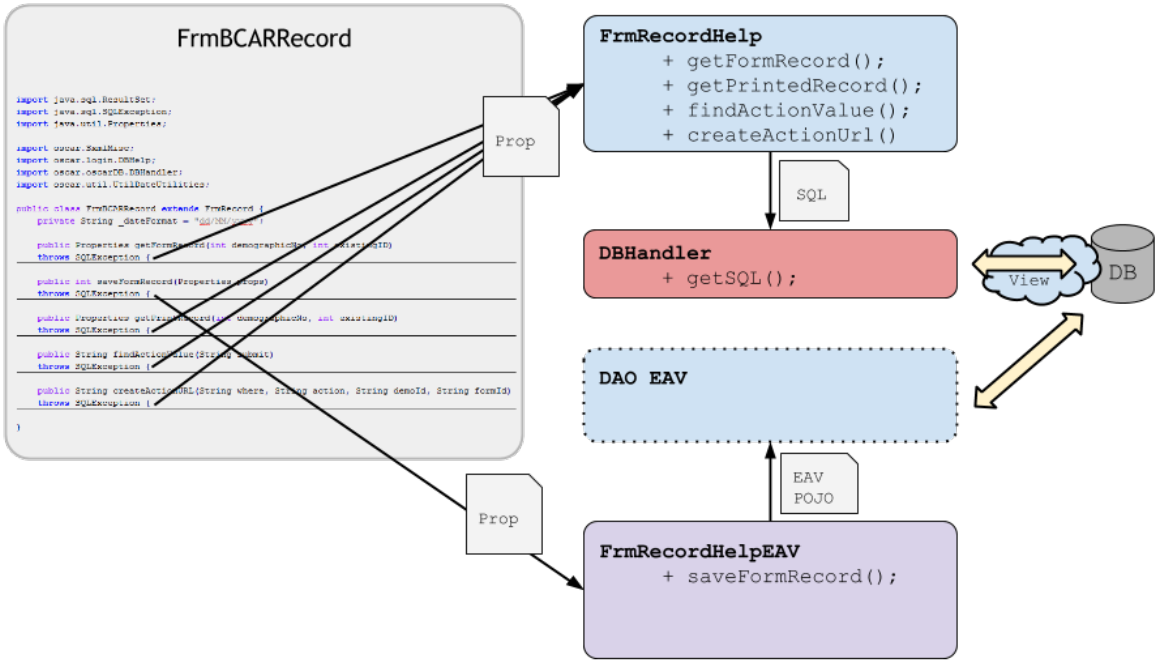


Figure 7.3 – Data Access Layer architecture after factoring

7.4 Code-refactoring impacts

In this chapter, we presented a number of MySQL limitations and their impacts on how we made the application evolve to face them. In fact, a lot of concepts that are presented in previous chapters cannot be used in the real world, at least not using MySQL.

The important point is that the refactoring was considerably eased by the previous work. Considering the time we spent on the code refactoring, we estimate that the views definitions removed 80% of the effort, allowing the program to run without modification in most cases.

The only work that has been done was to provide code for updating the EAV model directly from the application, without any triggers. Considering of the number of forms (about 70) in the BC OSCAR release and the deeply ingrained link between the associated classes and the original tables, providing views was the only viable solution to provide a functioning solution in a relatively short period of time.

Also, the OSCAR community is now thinking about migrating their DMBS to another, partly due to previously presented limitations. Therefore, the designed solution could potentially solve this migration problem as an *ad-hoc* solution, leaving the OSCAR software code almost unchanged.

Chapter 8

Conclusions

In the first pages of this work, we presented the purpose of this thesis. We explained why system evolution is an important topic and in the case study, emphasised this by presenting how our contributions to the OSCAR database schema evolution could help integrating data into data mining applications which aim to improve patient cares.

In the different chapters, we analyzed how database evolution raises the challenge of co-evolving all program code that uses the database unless we can implement "adapters" that allow programs to remain unchanged and use the database in its "old format".

In Chapter 2, we introduced the concepts of bidirectional transformations (bx) and coupled-transformations. We also presented how possible theories and tools can play an important role in keeping legacy applications running while evolving the database to a more suitable structure.

In Chapter 3, we spoke about data sparseness of the system and proposed another structure to store information. We then formalized the implementation of the Channels and presented a loss-less and type-safe migration process to the previously designed data structure. We also proposed several alternatives for synchronising the database structure with the views.

In Chapter 4, we provided a concrete implementation of the previously designed transformations and we have reported on experiences of generating Channels "at scale".

In Chapter 5, we presented a tool developed to facilitate this migration process aiming at ensuring *scalability* and *correctness* by implementing solutions presented in Chapter 3 and Chapter 4.

Finally, we contextualized our work by presenting a case study discussion and how we applied our techniques on this large and complex real-world system. To date, implementation alternatives of the transformations used in this work have not been studied at scale. We present performance and scalability aspects related to different implementation techniques.

Through this study, we answered the different questions. We summarize here what we have found at the end of this study.

Is it possible to implement co-evolution with bidirectionally concepts "At scale" ?

Through this work, we have presented the theoretical concepts of bidirectionality and we have spoken about current existing implementations. To date, we have not found any *ad-hoc* solution directly applicable for our specific case. We have implemented a solution, which despite its limitations, is perfectly applicable in a production system.

What are the limitations of such a possible implementation ?

We have seen that, depending on the DBMS considered, several limitations may occur. While most commercial implementations have all the required constructions needed for this work (updatable views, triggers on views, etc.), some others may not implement them. Therefore, little adaptations may be required.

Can the migration be loss-less and semantics-preserving ?

By the means of Channels and *bx* transformations, we showed that implementing *loss-less* and *semantics-preserving* transformations was feasible and also a keystone in our specific case study.

One of the most rewarding conclusions of this work is probably also the request from several people to use our tool in order to face similar or related real life problems. Now that we provided a way to refactor problematic tables into another data structure, we encourage the next interns who will finally take care of integrating OSCAR EMR data into the data mining system.

8.1 Additional discussion

During our research on how to define and implement Channels, a lot of ideas came to us. Some of these ideas came from us, and others from people that gave us feedback and advice. Although we tried to implement most of these idea, we had a time restriction and as a result, we did not had the opportunity to explore some interesting ways.

Firstly, several people came to us, facing several common problems and there are many other systems where our solution could potentially be applied. Working on other case studies would probably impact our solution, improving it and making it even more generic.

We also did not had the opportunity to research ways in which Channel transformations could be implemented by means of object-relational mapping descriptions. Current object-relational middleware does not have support for complex transformations, such as Pivot and Unpivot, and would have to be extended to implement such Channels. Providing support for those kinds of complex transformations could be interesting for the *bx* community.

Bibliography

[1] *JAVA INTERFACE FOR DB-MAIN - REFERENCE MANUAL*, February 2012.

[2] J. Bisbal-J. Grimson V. Wade B. Wu, D. Lawless and R. Richardson. Legacy system migration : A legacy data migration engine. PODS '06, New York, NY, USA, 1997.

[3] F. Bancilhon and N. Spyratos. Update semantics of relational views. *ACM Trans. Database Syst.*, 6(4):557–575, December 1981.

[4] Keith Bennett. Legacy systems: Coping with success. *IEEE Softw.*, 12(1):19–23, January 1995.

[5] Keith H. Bennett and Václav T. Rajlich. Software maintenance and evolution: A roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 73–87, New York, NY, USA, 2000. ACM.

[6] Jesús Bisbal, Deirdre Lawless, Bing Wu, and Jane Grimson. Legacy information systems: Issues and directions. *IEEE Softw.*, 16(5):103–111, September 1999.

[7] Aaron Bohannon, Benjamin C. Pierce, and Jeffrey A. Vaughan. Relational lenses: A language for updatable views. In *Proceedings of the Twenty-fifth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '06, pages 338–347, New York, NY, USA, 2006. ACM.

[8] Michael L. Brodie and Michael Stonebraker. *Migrating Legacy Systems. Gateways, Interfaces, and the Incremental Approach.* Morgan Kaufmann, 1995.

[9] John Corwin, Perry Miller, Avi Silberschatz, and Luis Marenco. Dynamic tables: An architecture for managing evolving, heterogeneous biomedical data in relational database management systems. *Journal of the American Medical Informatics Association*, 14:2007, 2007.

[10] Carlo A. Curino, Hyun J. Moon, and Carlo Zaniolo. Graceful database schema evolution: The prism workbench. *Proc. VLDB Endow.*, 1(1):761–772, August 2008.

[11] Krzysztof Czarnecki, J. Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James F. Terwilliger. Bidirectional transformations: A cross-discipline perspective. In *Proceedings of the 2Nd International Conference on Theory and Practice of Model Transformations*, ICMT '09, pages 260–283, Berlin, Heidelberg, 2009. Springer-Verlag.

[12] Umeshwar Dayal and Philip A. Bernstein. On the correct translation of update operations on relational views. *ACM Trans. Database Syst.*, 7(3):381–416, September 1982.

[13] Alin Deutsch and Val Tannen. Mars: A system for publishing xml from mixed and redundant storage. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, VLDB '03, pages 201–212. VLDB Endowment, 2003.

[14] Valentin Dinu and Prakash Nadkarni. Guidelines for the effective use of entity-attribute-value modeling for biomedical databases. *International Journal of Medical Informatics*, In Press, Corrected Proof:1056, 2006.

[15] Valentin Dinu, Prakash Nadkarni, and Cynthia Brandt. Pivoting approaches for bulk extraction of entity-attribute-value data. *Comput. Methods Prog. Biomed.*, 82(1):38–43, April 2006.

[16] Ronald Fagin. Inverting schema mappings. In *Proceedings of the Twenty-fifth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '06, pages 50–59, New York, NY, USA, 2006. ACM.

[17] Ronald Fagin, Phokion G. Kolaitis, Lucian Popa, and Wang Chiew Tan. Quasi-inverses of schema mappings. *ACM Trans. Database Syst.*, 33(2), 2008.

[18] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bi-directional tree transformations: A linguistic approach to the view update problem. *SIGPLAN Not.*, 40(1):233–246, January 2005.

[19] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2 edition, 2008.

[20] Jean-Luc Hainaut. In Ralf Lämmel, João Saraiva, and Joost Visser, editors, *Generative and Transformational Techniques in Software Engineering*, Lecture Notes in Computer Science, pages 95–143. Springer-Verlag.

[21] Jean-Luc Hainaut, Anthony Cleve, Jean Henrard, and Jean-Marc Hick. Migration of legacy information systems. In *Software Evolution*, pages 105–138. Springer Berlin Heidelberg, 2008.

[22] Cleve A. Hainaut J. Henrard, J. Inverse wrappers for legacy information systems migration. pages 30–43, 2004.

[23] J. Henrard, J-M. Hick, P. Thiran, and J-L. Hainaut. Strategies for data reengineering. In *Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02)*, WCRE '02, pages 211–, Washington, DC, USA, 2002. IEEE Computer Society.

[24] IEEE. *Standard IEEE Std 1219-1999 on Software Maintenance*, volume 2. IEEE Press, 1999.

[25] Mira Kajko-Mattsson. The state of documentation practice within corrective maintenance. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, ICSM '01, pages 354–, Washington, DC, USA, 2001. IEEE Computer Society.

[26] Bill Karwin. *SQL Antipatterns : Avoiding the Pitfalls of Database Programming*. PRAGMATIC BOOKSHELF, 2010.

[27] Ralf Lämmel. Coupled software transformations. *First international workshop on software evolution transformations*, pages 31–35, 2004.

[28] M. M. Lehman. Laws of software evolution revisited. In *Proceedings of the 5th European Workshop on Software Process Technology*, EWSPT '96, pages 108–124, London, UK, UK, 1996. Springer-Verlag.

[29] Meir M. Lehman. Programs, life cycles, and laws of software evolution. *Proc. IEEE*, 68(9):1060–1076, September 1980.

[30] Ingo Lütkebohle. OSCAR: CONNECTING CARE, CREATING COMMUNITY — OSCAR Canada Users Society. http://oscarcanada.org/. [Online; accessed 08-August-2014].

[31] Anthony Cleve-Jens Weber Maxime Gobert, Jérôme Maes. Understanding schema evolution as a basis for database reengineering. *icsm2013*, 2013.

[32] Sergey Melnik, Atul Adya, and Philip A. Bernstein. Compiling mappings to bridge applications and databases. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, pages 461–472, New York, NY, USA, 2007. ACM.

[33] Robert Orfali, Dan Harkey, and Jeri Edwards. *The Essential Distributed Objects Survival Guide*. John Wiley & Sons, Inc., New York, NY, USA, 1995.

[34] A. M. Riad Mohammed Elmogy Shaker H. El-Sappagh, Samir El-Masri. Electronic health record data model optimized for knowledge discovery. *IJCSI International Journal of Computer Science Issues*, 9:329, 2012.

[35] Harry M. Sneed. *Objektorientierte Softwaremigration: [praxiserprobtes Konzept für die Migration von Legacy-Systemen ; Sanierung der Oberflächen, Daten und Programme ; Konversion von 3-GL-Programmen in oo-Programme ; Kapselung alter Programme und Datenbanken]*. Professionelle Softwareentwicklung. Adddison-Wesley Longman, Bonn, 1. aufl. edition, 1999.

[36] E. Burton Swanson. The dimensions of maintenance. In *Proceedings of the 2Nd International Conference on Software Engineering*, ICSE '76, pages 492–497, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[37] James F. Terwilliger. Bidirectional by necessity: Data persistence and adaptability for evolving application development. In *GTTSE*, pages 219–270, 2011.

[38] James F. Terwilliger. Bidirectional by necessity: Data persistence and adaptability for evolving application development. In Ralf Lämmel, João Saraiva, and Joost Visser, editors, *Generative and Transformational Techniques in Software Engineering IV*, volume 7680 of *Lecture Notes in Computer Science*, pages 219–270. Springer Berlin Heidelberg, 2013.

[39] James F. Terwilliger, Anthony Cleve, and Carlo Curino. In Zhenjiang Hu and Juan de Lara, editors, *ICMT*, Lecture Notes in Computer Science, pages 1–23. Springer.

[40] James F. Terwilliger, Anthony Cleve, and Carlo Curino. How clean is your sandbox? - towards a unified theoretical framework for incremental bidirectional transformations. In *ICMT*, pages 1–23, June 2012.

[41] James F. Terwilliger, Lois M. L. Delcambre, David Maier, Jeremy Steinhauer, and Scott Britell. Updatable and evolvable transforms for virtual databases. *Proc. VLDB Endow.*, 3(1-2):309–319, September 2010.

[42] Philippe Thiran, Jean-Luc Hainaut, and Geert-Jan Houben. Database wrappers development: Towards automatic generation. In *CSMR*, pages 207–216. IEEE Computer Society, 2005.

[43] Philippe Thiran, Jean-Luc Hainaut, Geert-Jan Houben, and Djamal Benslimane. Wrapper-based evolution of legacy information systems. *ACM Trans. Softw. Eng. Methodol.*, 15(4):329–359, October 2006.

[44] Philippe Thiran, Jean-Marc Hick, and Jean-Luc Hainaut. Generation of conceptual wrappers for legacy databases. In Trevor J. M. Bench-Capon, Giovanni Soda, and A. Min Tjoa, editors, *DEXA*, Lecture Notes in Computer Science, pages 678–687. Springer.

[45] S.R. Tilley and D. Smith. Perspectives on legacy system reengineering, 1995.

[46] Catharine M. Wyss and Edward L. Robertson. A formal characterization of pivot/unpivot. In *Proceedings of the 14th ACM International Conference on Information and Knowledge Management*, CIKM '05, pages 602–608, New York, NY, USA, 2005. ACM.